
AFRL-RY-WP-TR-2016-0078

# HARDWARE ACCELERATION OF SPARSE COGNITIVE ALGORITHMS

**Paul D Franzon, Lee Baker, Sumon Dey, Weifu Li, and Joshua Schabel**

**North Carolina State University**

**MAY 2016**
**Final Report**

**Approved for public release; distribution unlimited.**

***See additional restrictions described on inside pages***

**STINFO COPY**

**© 2016 Paul D Franzon, Lee Baker, Sumon Dey, Weifu Li, and Joshua Schabel**

**AIR FORCE RESEARCH LABORATORY**
**SENSORS DIRECTORATE**
**WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320**
**AIR FORCE MATERIEL COMMAND**
**UNITED STATES AIR FORCE**

# NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2016-0078 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

_//Signature//_____
ALFRED J. SCARPELLI, Program Manager
Advanced Sensor Components Branch
Aerospace Component & Subsystems Division

//Signature//_____
BRADLEY PAUL, Chief
Advanced Sensor Components Branch
Aerospace Component & Subsystems Division

_//Signature//_____
F. JESSE FANNING, Chief
Aerospace Component & Subsystems Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS**.

| 1. REPORT DATE *(DD-MM-YY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| May 2016 | Final | 01 December 2014 – 15 February 2016 |

**4. TITLE AND SUBTITLE**
HARDWARE ACCELERATION OF SPARSE COGNITIVE ALGORITHMS

**5a. CONTRACT NUMBER**
FA8650-15-1-7518

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**
61101E

**6. AUTHOR(S)**
Paul D Franzon, Lee Baker, Sumon Dey, Weifu Li, and Joshua Schabel

**5d. PROJECT NUMBER**
1000

**5e. TASK NUMBER**
N/A

**5f. WORK UNIT NUMBER**
Y18M

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

North Carolina State University
2701 Sullivan Drive
Suite 240, Campus Box 7514
Raleigh, NC 27695-7003

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency
DARPA/MTO
675 North Randolph Street
Arlington, VA 22203

**10. SPONSORING/MONITORING AGENCY ACRONYM(S)**
AFRL/RYDI

**11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)**
AFRL-RY-WP-TR-2016-0078

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited.

**13. SUPPLEMENTARY NOTES**
This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This material is based on research sponsored by Air Force Research Laboratory (AFRL) and the Defense Advanced Research Agency (DARPA) under agreement number FA8650-15-1-7518. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and the Defense Advanced Research Agency (DARPA) or the U.S. Government. © 2016 Paul D Franzon, Lee Baker, Sumon Dey, Weifu Li, and Joshua Schabel. This work was funded in whole or in part by Department of the Air Force Grant FA8650-15-1-7518. The U.S. Government has for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable worldwide license to use, modify, reproduce, release, perform, display, or disclose the work by or on behalf of the U. S. Government. Report contains color.

**14. ABSTRACT**

Hardware accelerators were designed for both the Sparsey and Numenta HTM cortical algorithms. Two versions were designed – a programmable 65 nm SIMD version with Processor in Memory (PiM) extensions and a 65 nm ASIC version. They were compared against a 28 nm GPU baseline using the KTH video action recognition benchmark. Performance/power improvement over the GPU were (Sparsey) SIMD with PiM: 1490; ASIC: 1300; and (HTM) SIMD with PiM: 537; ASIC: 47,100.

**15. SUBJECT TERMS**
Cortical Algorithms; Machine Learning; Hardware; VLSI; ASIC

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT: | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON (Monitor) |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Alfred J. Scarpelli |
| Unclassified | Unclassified | Unclassified | SAR | 46 | 19b. TELEPHONE NUMBER *(Include Area Code)* N/A |

# Table of Contents

**Section** **Page**




Approved for public release; distribution unlimited.

# List of Figures

**Figure** **Page**




Approved for public release; distribution unlimited.

# List of Tables

**Table** **Page**




Approved for public release; distribution unlimited.

# 1.0 EXECUTIVE SUMMARY

The goal of this project was to determine the potential for improvement of performance per unit of power for customized implementations of the Sparsey and Numenta Hierarchical Temporal Memory (HTM) algorithms as compared with a graphics processing unit (GPU) baseline, using the KTH Royal Institute of Technology (KTH) action recognition benchmark.

The overall architecture of the customized implementation is shown in Figure 1. The computation is performed in a processing element (PE). Three different PEs are being investigated:

1. A programmable single instruction multiple data (SIMD) solution. The SIMD solution instruction set is customized to these applications, while retaining general purpose utility.
2. A parameterized design for Sparsey. I.e. this is an application-specific integrated circuit (ASIC) in which certain parameters can be changed at compile time and/or run time. However, it can only run one version of the Sparsey algorithm.
3. A parameterized design for Numenta HTM. I.e. this is an ASIC in which certain parameters can be changed at compile time and/or run time. However, it can only run one version of the Numenta HTM algorithm.

For the ASICs, each PE has its own local memory, built as a static random access memory (SRAM). The SIMD PEs are grouped into clusters and share memory at the cluster level. The SIMD PEs includes both special function units (SFUs) and process-in-memory (PIM) accelerators. The SFUs are attached to the register file ports of the execution lanes while the PIMs are connected to bank(s) of memory. The SFU and PIM accelerators deliver an order of magnitude speedup over the SIMD PEs alone for the hot spots of the algorithms. A SIMD primitive includes a circuit switched network to connect multiple clusters together to global (dynamic random access memory (DRAM)) memory.

The SIMD architecture has been implemented at the register transfer language (RTL) level, then synthesized and routed on a commercial 65 nm library to obtain more accurate area, performance, and power predictions.

The results obtained are shown in Table. 1. In general, the programmable SIMD solution achieved an improvement in performance/power in the range of three to four orders of magnitude. Most of this improvement was due to the PIM enhancements introduced. These are direct vector logic operations wherein the operands and results are in the main memory (SRAM here). Without the PIM, the improvement for performance/power over the GPU was just - 50 for Sparsey and 136 for HTM, vs. 1490 and 537 with the PIM features.

The improvement was larger in Sparsey than in HTM, as the former is more regular in nature and thus more vectorizable. The ASIC accelerator for Sparsey achieved a 1300x improvement in performance/power over the baseline. Key innovations in this ASIC include distributed memory, and reduced precision floating point. However, its improvement would exceed $10^4$ if the PIM components were introduced into the ASIC architecture. Time did not permit that. The ASIC accelerator for HTM, achieved an improvement in performance/power of 47,100. Key


Approved for public release; distribution unlimited.

innovations that led to this result included coordinated, micro-parallelism, especially in the sort stages, and a distributed memory architecture.

In general, these algorithms are more memory bound than compute bound. They require much larger working memories, and more memory bandwidth, than conventional "deep learning" networks, especially when the latter are pruned. Hence the high value of the PIM and other innovations that help reduce the memory wall.

These ASICs were implemented with SRAMs. Capacity dictates that it would be preferable to use DRAMs, or storage class memories (SCM), such as NAND flash, or eventually phase change memory (PCM) or resistive random access memory (RRAM). Of particularly interest is the potential to perform the PIM operations on an evolved 3D DRAM, such as the 15 ns 4+ Tbps Tezzaron DiRAM.

**Table 1. Summary of Demonstrated Results**

*Notes: (1) Server class GPU, NVidia Tesla K20c. (2) Consumer grade GPU, GeForce GT 640.*

| Factor | GPU Baseline | SIMD with PiM | ASIC |
|---|---|---|---|
| **Sparsey** | | | |
| KTH Run time | 8 ms (1) | 0.53 ms | 0.06 ms |
| Power | 200 W | 2.025 W | 22 W |
| Logic Area | | 26 $mm^2$ | 220 $mm^2$ |
| Performance/Power | 1 | 1490 | 1300 |
| **Numenta HTM** | | | |
| KTH run time | 225 ms (2) | 12.7 ms | 0.63 ms |
| Power | 70 W | 2.308 W | 298mW |
| Logic Area | | 26 $mm^2$ | 5.5$mm^2$ |
| Performance/Power | 1 | 537 | 47,100 |

In conclusion, it is clear that these emerging algorithms that can support unsupervised, or lightly supervised learning, as well as incremental learning, map poorly onto conventional computing architectures. The reason is that core to these algorithms, everything is "remembered", and the resulting reinforced weights cannot be optimized down to a smaller set via off-line optimization.

As a result, this study showed that custom hardware, implemented in the 65 nm node, can improve the power/performance ratio compared with 40 nm/28 nm GPUs by a factor of 100,000 or more. The main reason for this is that the custom solutions have more useable memory bandwidth, and are designed to exploit it. Key innovations demonstrated include the use of PIM, and the design of parallelized micro-op sequences that permitted faster overall operation.


Approved for public release; distribution unlimited.

# 2.0 INTRODUCTION

The goal of this project is to determine the potential for improvement of performance per unit of power for customized implementations of the Sparsey and Numenta HTM algorithms as compared with a GPU baseline.

The overall architecture of the customized implementation is shown in Figure 1.1. The computation is performed in a PE. Three different PEs are being investigated:

1. A programmable SIMD solution. The SIMD solution instruction set is customized to these applications, while retaining general purpose utility.
2. A parameterized design for Sparsey.
3. A parameterized design for Numenta HTM.

Each PE has its own local memory, built as an SRAM. The SIMD PE also has a PIM unit that accelerates certain distributed operations. The PIM unit has delivered an order of magnitude speedup over the SIMD PE alone. A cluster of PEs are connected via a local circuit switched network. A global network provides connections to other clusters and to the global (DRAM) memory.

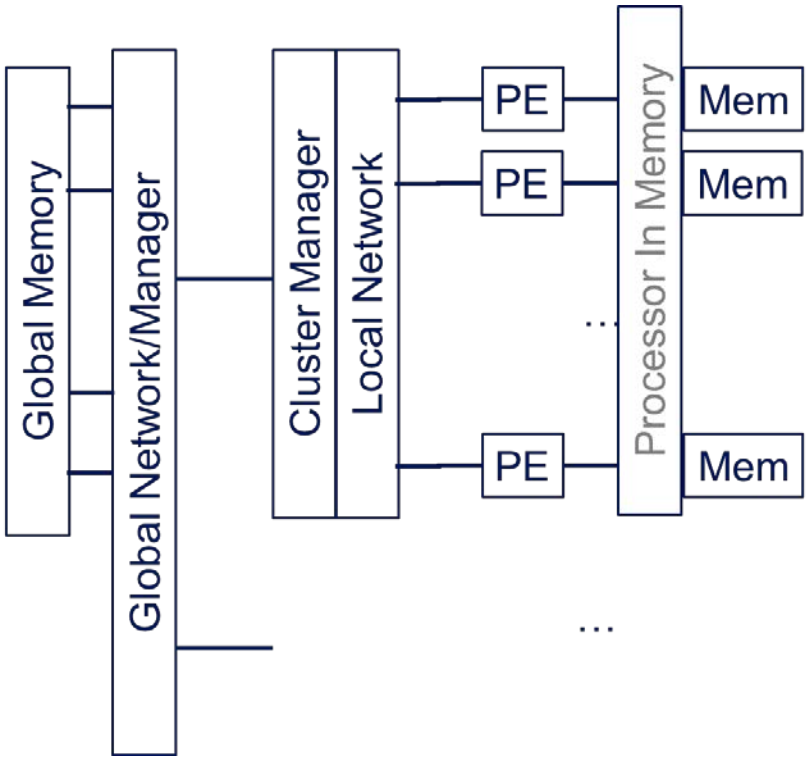


**Figure 1: Overall Architecture**


Approved for public release; distribution unlimited.

In general the approach taken to satisfy the requirements for this effort were as follows:

1. GPU code was written for both a version of Sparsey and HTM. This was run on a GPU to develop the baseline for GPU performance. The KTH benchmark was used. [KTH]
2. A design was generated either at the synthesizable RTL or transaction level model (TLM) levels. All logic was implemented at the RTL level so that area, speed and power could be estimated accurately. These models can be simulated to determine the number of clock cycles required to complete a task.
3. The RTL was synthesized and placed and routed using either Global Foundries (GF) 130 nm or GF 65 nm libraries. This gives the cell area, the achievable clock period and a netlist. Scaling of the results to the 28 nm node was also calculated. Power and performance are calculated based on synthesized results while running the benchmark.
4. The results were interpreted for trends.

The report is structured as follows. Section 2 summarizes the GPU software only results. Then the three implementation studies we conducted are explained: the SIMD solution, and each of the Sparsey and HTM ASIC implementations. These sections are broken into implementation details, key results and a discussion of relevant factors, such as precision needed and issues involved in scaling to larger problems. Finally some 3D implementation issues are discussed before concluding.


Approved for public release; distribution unlimited.

# 3.0 GPU BENCHMARKING

## 3.1 Sparsey

Although overall system run time was improved, this overall time includes various system overheads that may not be required if a GPU were used in an embedded system. Therefore, it was decided to observe run-times of individual kernels. The kernel run times for all layers for an individual input can be seen in Figure 2 Kernel run times on an NVidia GTX750 using the NVidia profiler.

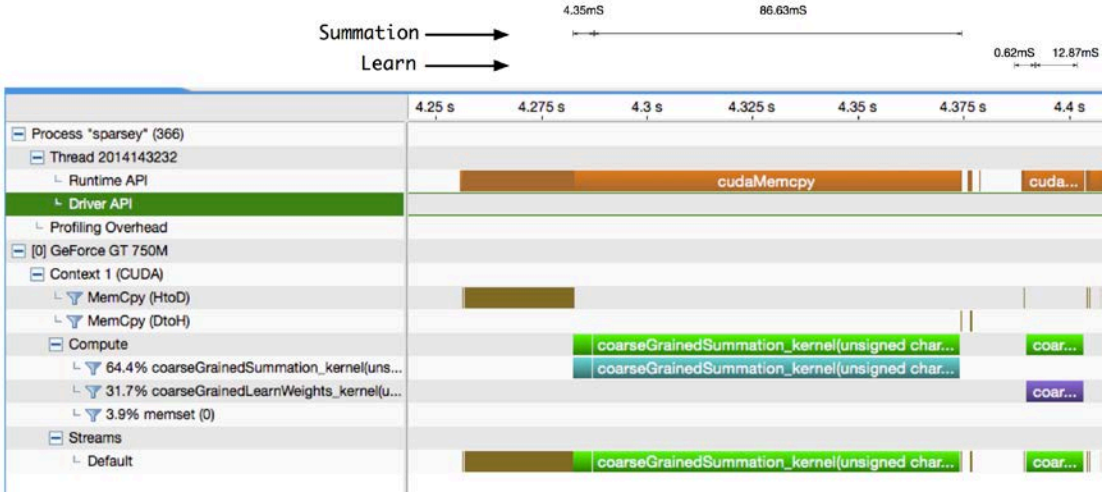


**Figure 2: Kernel Run Times on an NVidia GTX750 Using the NVidia Profiler**

In this example shown in Figure 2, a GTX750 is being employed. The GTX750 is a "mobile" class GPU. In this case, the Kernel times taken for the bottom-up and horizontal summation times are 4.35mS and 86.63mS respectively.

These same kernels were run on a "server" class GPU running on an NVidia Tesla K20c. The times taken for the bottom-up and horizontal summation times were 0.39mS and 7.83mS respectively.

The various kernel run times for both these GPU's will be used for future comparisons avoiding any overhead associated with data transfers to and from the host CPU.

Power was calculated using datasheet values.

## 3.2 HTM

The execution time for each kernel running on GeForce GT 640 is summarized in Table 2.

Totally 240 images are used as input files for both learning phase and inference phase. Since various system overheads, such as data transfer between host and device may not be necessary if the GPU is used in an embedded system, only execution time will be used for future comparisons.


Approved for public release; distribution unlimited.

**Table 2. Kernel Execution Time**

| Kernel Function | Execution Time Per Image (*ms*) |
|---|---|
| Kernel_GetOverlap() | 0.62 |
| Kernel_CreateSegment() | 190.76 |
| Kernel_Inhibit() | 5.53 |

Though the overall simulation result of GPU implementation doesn't perform a significant improvement, the execution time of each hotpot is dramatically improved, which demonstrates the necessity of further exploring the parallelism in HTM algorithm on custom ASIC.

Power was calculated from datasheet values.


Approved for public release; distribution unlimited.

# 4.0 PROGRAMMABLE SIMD SOLUTION

## 4.1 Architecture and Design

Figure 3 shows the SIMD node in a 2D implementation in which the instruction and data scratch pad memories (SRAMs in this work) are placed on the same die. Figure 4 shows the SIMD node in a 3D implementation in which the scratch pad memories implemented on a separate die with the PIM accelerators. A cluster in the SIMD architecture is defined as 2+ nodes and a control core communicating either through a shared memory or a simple ring or crossbar interconnect. Figure 5 shows the 3D implementation of a cluster with eight processing nodes that communicate via a stacked shared memory with PIMs integrated in between the banks. A galaxy is defined as 2+ clusters and a control core communicating via either a ring-based interconnect or a crossbar. Figure 6 shows a block diagram of a galaxy with four clusters, which is the defined SIMD primitive in this work. The galaxy level is also where access to main memory (such as DRAM) would occur.

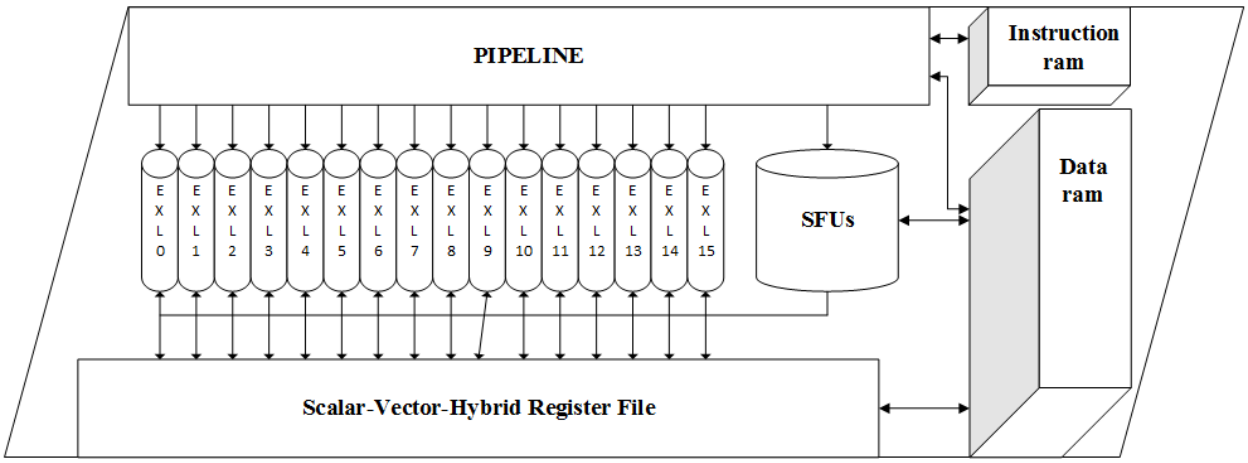


**Figure 3: 2D Implementation of SIMD Node**


Approved for public release; distribution unlimited.

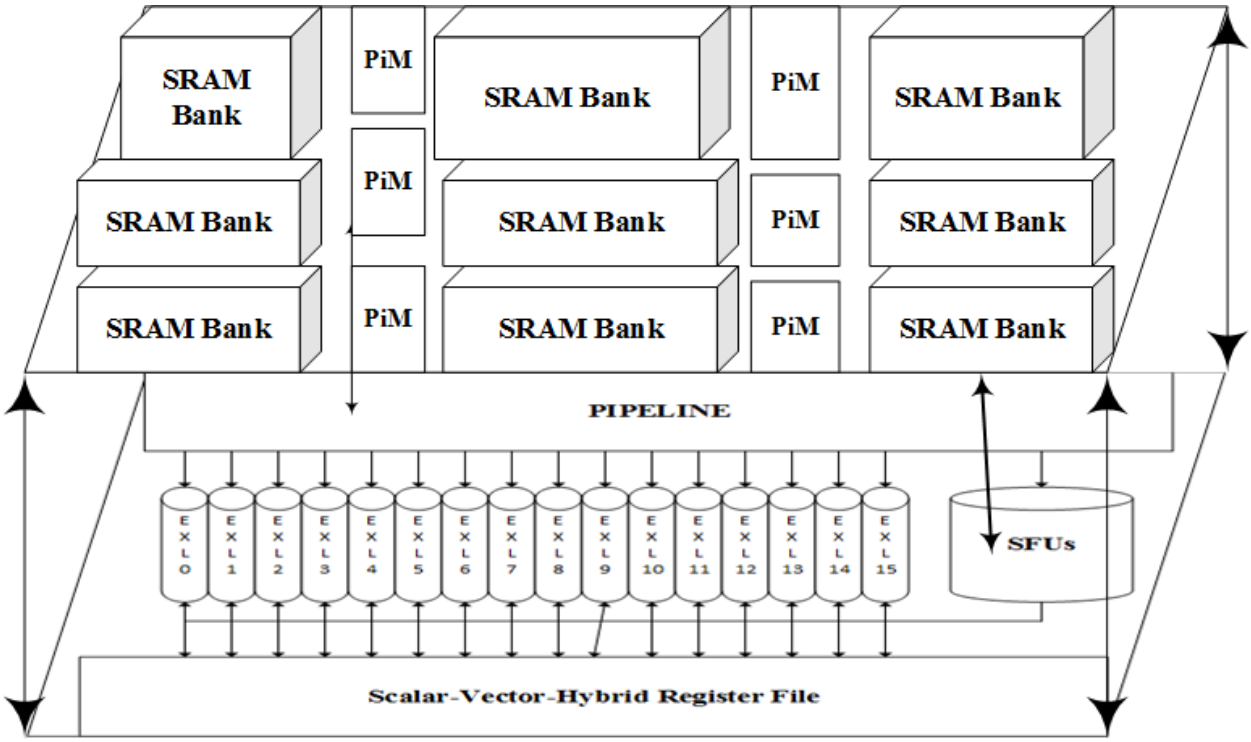


**Figure 4: 3D Implementation of SIMD Node**

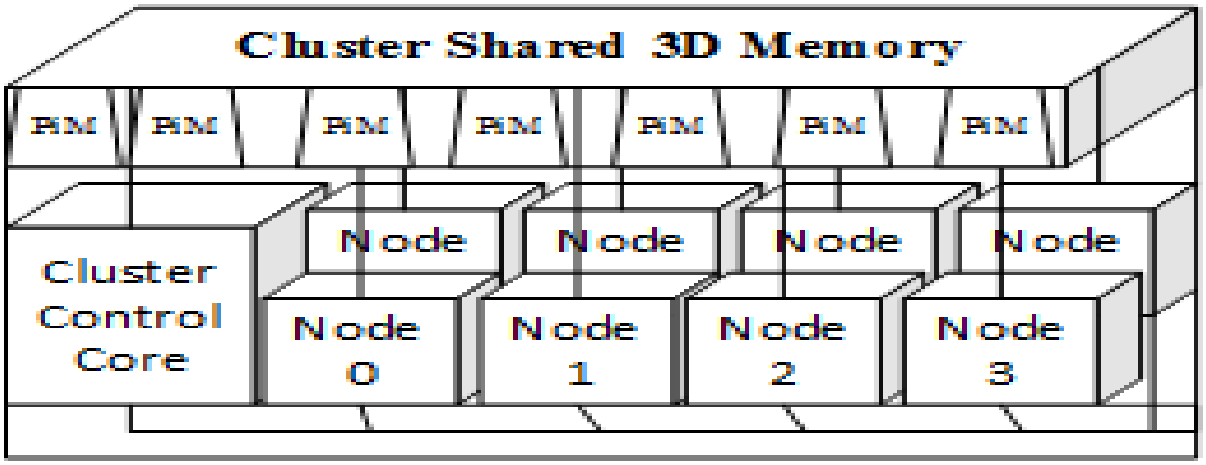


**Figure 5: 3D Implementation of a Cluster**


Approved for public release; distribution unlimited.

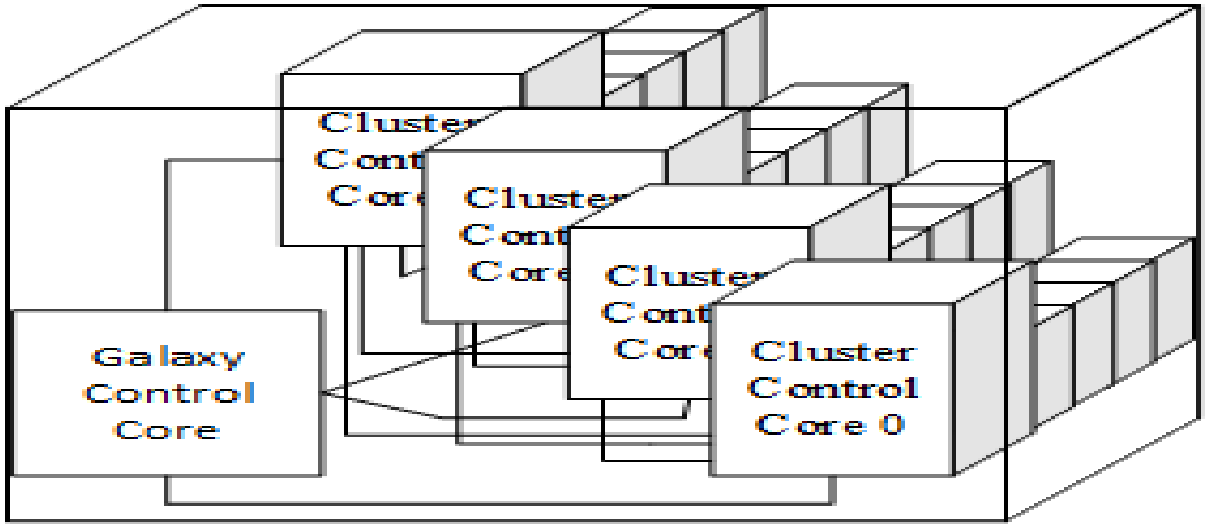


**Figure 6: Galaxy Block Diagram**

The remainder of this section outlines the extensions to the SIMD node instruction set architecture (ISA) for cortical operations (HTM and Sparsey), including calling the PIMs and SFUs. The SFUs are implemented next to the execution lanes and utilize the register file ports to the execution lanes to perform scalar operations on vectored data, such as summations.

**Sparsey:**

- BSUM
- I2F –
- FMAX
- FMIN
- ONEHOT
- PIM_SELECT_WINNER_LD
- PIM_SELECT_WINNER_LE
- PIM_UPDATE_WINNER
- PIM_WEIGHT_SUM
- I2F

**HTM:**

- IMAX
- IMIN
- GETBIT
- ICMP
- SFU_VECTOR_SUMMATION
- SFU_HTM_CREATE_BIT_VECTOR
- PIM_HTM_SCATTER_GATHER
- PIM_NODE_BITONIC_SORT_FP
- PIM_BIDE_BITONIC_SORT_INT
- PIM_CLUSTER_MULTI_MERGE
- PIM_GALAXY_MULTI_MERGE
- LDI or STI to address zero is null


Approved for public release; distribution unlimited.

## 4.2 Results

**Table 3. Performance Results for Sparsey on SIMD Architecture**

| Sparsey Implementation | Run Time Per Frame @ 250 MHz |
|---|---|
| Learning disabled on SIMD with no PIMs | 8.63 ms |
| Learning disabled on SIMD with PIM for weight summation step only | 853 µs |
| Learning disabled on SIMD with PIMs for weight summation and select winner steps | 389 µs |
| Learning enabled on SIMD with no PIMs | 9.26 ms |
| Learning enabled on SIMD with PIM for weight summation step only | 1.49 ms |
| Learning enabled on SIMD with PIMs for weight summation and update winner steps | 713 µs |
| Learning enabled on SIMD with PIMs for weight summation, select winner, and update winner steps | 523 µs |

**Table 4. Performance Results for HTM on SIMD Architecture**

| HTM Implementation | Run Time Per Frame @ 250 MHz |
|---|---|
| Entirely on SIMD with only node-level bitonic sort PIMs for selecting active columns and sorting active columns by Euclidean distance | 51 ms |
| Entirely on SIMD with node-level bitonic sort PIMs plus scatter gather PIM | 15.7 ms |
| Entirely on SIMD with node-level bitonic sort PIMs plus scatter gather PIM and bit vector shift SFU | 12.7 ms |

**Table 5. Area Results of SIMD Architecture at 65 nm**

| Processing Element | Area |
|---|---|
| SIMD with cortical extensions for in-lane execution functions | 0.9 $mm^2$ at 130 nm for 250 MHz<br>0.51 $mm^2$ at 65 nm scaled conservatively |
| SIMD with cortical extensions for in-lane execution functions, SFUs and PIMs | 0.69 $mm^2$ at 65 nm for 100 MHz |
| HTM scatter gather SFU/PIM | 0.01 $mm^2$ at 65 nm for 250 MHz |
| Create bit vector SFU | 0.001 $mm^2$ at 65 nm for 250 MHz |
| Vector summation SFU | 0.006 $mm^2$ at 65 nm for 250 MHz |
| Sparsey weight summation PIM | 0.002 $mm^2$ at 65 nm for 250 MHz |
| Sparsey update winner PIM | 0.002 $mm^2$ at 65 nm for 250 MHz |
| Sparsey select winner LE PIM | 0.003 $mm^2$ at 65 nm for 250 MHz |
| Sparsey select winner LD PiM | 0.003 $mm^2$ at 65 nm for 250 MHz |
| HTM 64 element integer bitonic sort PIM | 0.077 $mm^2$ at 65 nm for 250 MHz |
| HTM 32 element FP bitonic sort PIM | 0.038 $mm^2$ at 65 nm for 250 MHz |

**Table 6. Power Results at 65 nm, 250 MHz**

| Processing Element | Average Power |
|---|---|
| Single 16-lane PE running Sparsey (no PIMs)* | 62 mW |
| 32 16-lane Primitive running Sparsey (no PIMs)* | 2 W + 25 mW for memory access |
| 32 16-lane Primitive running HTM (no PIMs) | 2.3 W + 8.19 mW for memory access |
| Sparsey weight summation PIM | 264 µW |
| Sparsey update winner PIM | 313 µW |
| Sparsey select winner LE PIM | 192 µW |
| Sparsey select winner LD PIM | 192 µW |
| HTM 64 element integer bitonic sort PIM | 3 mW |
| HTM 32 element FP bitonic sort PIM | 5 mW |

*scaled from 130 nm

**Table 7. Sparsey Communications on SIMD Primitive**

| | # bits to transmit per frame | |
|---|---|---|
| | **point 2 point** | **broadcast** |
| L0 H-input | 26624 | 24576 |
| L1 H-input | 10240 | 8192 |
| L2 H-input | 1600 | 1600 |
| L0 to L1 BU input | 0 | 0 |
| L1 to L2 BU input | 24576 | 8192 |
| Output | 1600 | 1600 |
| Total | 64640 | 44160 |
| | **Bandwidth (Gbps)** | |
| | **point 2 point** | **broadcast** |
| L0 H-input | 64.64 | 44.16 |
| L1 H-input | 12.928 | 8.832 |
| L2 H-input | 6.464 | 4.416 |
| L0 to L1 BU input | 1.2928 | 0.8832 |
| L1 to L2 BU input | 0.6464 | 0.4416 |
| Output | 0.06464 | 0.04416 |

### 4.3 Discussion

SIMD Primitive: A cluster is comprised of eight SIMD nodes and cluster control core. The cluster shares a large memory, with embedded PIMs. Four clusters are connected with a control core to form a galaxy, which is also connected to the main memory.

Sparsey Baseline: Layer 0 is comprised of 16x10 macrocolumns (MACs), each MAC containing 32 competitive modules (CMs), and each CM containing 16 cells. Layer 1 is comprised of 4x4 MACs, each MAC containing 32 CMs, and each CM containing 16 cells. Layer 2 is comprised of 1 MAC with 100 CMs, 16 cells per CM.


Approved for public release; distribution unlimited.

HTM baseline: Single tier and single region implementation of 52x32 columns. Each column contains 48 proximal synapses connected to the input. Each column contains 32 cells. Each cell can dynamically form up to 16 distal segments. When a distal segment is created, it creates 16 distal synapses to cells in other currently active columns.

### 4.3.1 Mapping Sparsey to the SIMD Architecture

The focus of Sparsey is to calculate probabilities for selecting the current active cell in each CM in each MAC. This is achieved by summing the weights of each cell with the inputs to each MAC, normalizing the summations, using the max normalized summation of each CM to calculate a MAC's spatiotemporal familiarity, and finally using the MAC's spatiotemporal familiarity to determine each cell's probability of winning for the current frame. When learning is disabled, the cell with the largest probability is chosen as the winner and a uniformed random variable is utilized to induce sparsity when selecting the winner. When learning is enabled, a probability mass function is created for each cell in a CM. Sparsity is induced by selecting the winning cell with a probability mass function closest to a uniform random variable.

Sparsey is a "do all" algorithm. MACs and CMs are not considered active or inactive, only cells are, and since every cell could potentially become active for any given frame, all probability calculations must be performed. Sparsey synaptic connections are pre-known from network setup. For this work, we use a north-south-east-west neighboring for a MAC's synaptic connections to other MACs. Each frame, a MAC must form a new state vector, which is a single bit representing each cell within the MAC. It must transmit this state vector to itself, its horizontal neighbors, and its upward neighbor on the next layer. The only step that does only operate on active cells is the update weight step when learning is enabled.

Table 3 shows the performance results for the Sparsey base on a SIMD primitive for a clock frequency of 250 MHz. It is important to note that these results are not for the KTH algorithm, but randomized input. For Sparsey, this does not impact the implementation of the algorithm since Sparsey is a "do all" algorithm, and the only step that is skipped is updating the weights of the winning cells when learning is disabled.

When learning is disabled, the execution hot spots for Sparsey are weight summation and the select winner steps. The weight summation steps is a hot spot due to the large amount of data that must be moved from the SRAM to SIMD units, and the select winner step is hot spot due to the divergence embedded into the step that does not map well to a SIMD unit. Adding PIMs for the steps into the memory reduces the execution latency of a frame by 22X. When learning is enabled, weight update of the winning cells is added to the execution hot spot list. Adding PIMs for the execution hot spots reduces the execution latency by 17X. Table 3 does not include the time for transmitting the MAC states for horizontal and bottom-up inputs each frame between clusters.

Table 6 shows the average power running all Sparsey steps per frame on the SIMD architecture. The 16-lane SIMD tile has been optimally placed-and-routed on a commercial 130 nm process for a clock period of 300 MHz and area of 1.77 $mm^2$ (this area does not include any SFUs, PiMs, instruction scratch RAMs, or data scratch RAMs). The result presented in Table 6 has been conservatively scaled to 65 nm, where a 32 node SIMD primitive would consume an area of

36.75 $mm^2$ (excluding interconnect and assuming the cluster and galaxy control cores are also 16-lane units). If scaling conservatively to 32 nm from 130 nm, the SIMD primitive would be roughly 21 $mm^2$, operating at just under 1 W.

Table 7 shows the communications bandwidth required for the Sparsey baseline between clusters on a SIMD primitive. At 250 MHz, if 48-bit packets are used for communications, then 12 Gbps bandwidth is achievable and total communications can occur in less than 5 µs per frame.

### 4.3.2 Mapping HTM for the SIMD Architecture

HTM as described by the pseudo-code in the Numenta white sheet is not a highly parallel algorithm. Operations in terms of column-level, cell-level, segment-level, or synapse-level are parallelizable, but when implementing on a SIMD, the parallelization gets overrun by the high amount of divergence within the algorithm. For optimizing for SIMD, the algorithm has to be implemented for vectorization using predication and executing multiple paths. This can lead to significant increases in parallelization opportunities, but can be overwhelmed at the distal synapse level by the sheer amount of divergent paths and data that must be predicated for.

To account for this, the first attempt at mapping HTM involved breaking up all data elements out of the column-cell-segment-synapse data structures and looking at the data as mass arrays. The algorithm itself was then broken down into steps based on what data arrays were being operated on, and to find steps that were disjoint and could thus be assigned to different clusters and ran in parallel. This resulted in increases in parallelization, but also increases in execution latency and time required for communication of data between clusters. This was mainly due to early implementations utilizing local inhibition for creating new distal segments and synapses. The algorithm was readjusted to utilize global inhibition with some characteristics of local inhibition, which is directly related to how sparseness is handled.

To account for global inhibition, we divide the data arrays evenly among the number of processing nodes (52 columns per node). We set the max number of active columns for any given frame that can be active to 2% of the total number of columns (2% of 52x32 is 33, but 32 is used to remain in radix-2 with the number of nodes). In the spatial phase of HTM, the active columns are chosen based on their overlap values. If full global inhibition is used, then each node would determine its 32 columns with the largest overlap values and pass them to the cluster control core. The cluster control core must then determine the cluster's 32 columns with the largest overlap values and pass them to the galaxy control core. The galaxy control core then determines the entire network's active columns list and broadcasts the list to all nodes. Each node then evaluates the active columns list columns for columns that it is assigned, and transmits out the active, predict, and learn state vectors for the cells in the active columns. Most of the temporal phase of HTM operates only on active columns, and if each node has one active column, the temporal phase can be executed completely in parallel. Global inhibition does not guarantee each node would have an active column. So instead, local inhibition is enforced as well in that each cycle, each node selects its cell with the largest overlap value and sets it as active.

From Table 4, the SIMD primitive for HTM starts with PIMs only for two steps: in the spatial step for selecting the active columns, and the temporal phase, when creating a new distal segment on a cell of an active column, the Euclidean distance between that column and all other active columns must be calculated and sorted. Bitonic sort PIMs were implemented for these steps. This first implementation results in an execution latency of 51 ms (not including any data transmission times between clusters). Cluster-to-cluster communications are limited in this implementation of HTM on the SIMD primitive, with only 608 bits peak needing to be transferred and copied to all processing nodes.

There are four hot spots in this implementation. The first hot spot occurs in the spatial phase of HTM, when active columns have been chosen, and these active columns must look back at the previous active columns and build vectors that correlate to its distal synapses' connect cells in active columns. This step consumes 5.14 ms. This second hot spot also reflects the same issue in temporal phase, but accounts for all columns' distal synapses and their connection to cells in currently active columns. This step consumes 37.3 ms. Adding PIMs and vector unit SFUs to optimize these hot spots reduces the execution latency by 4X down to 12.7 ms. The third hot spot is also in the temporal phase, when calculating the segment active activity for setting the predict state of cells, and consumes 1.5 ms to execute. The final hot spot is at the end of the temporal phase is the final update to the permanence values of distal synapses, which requires 3.8 ms to execute.

Table 4 shows the performance results for HTM on a SIMD primitive for a clock frequency of 250 MHz. It is important to note that these results are not for the KTH algorithm, but randomized input (mainly testing functionality in simulation). Unlike with Sparsey, this simulation choice does change how HTM is simulated since it is a "dynamic algorithm," where the distal segments (and distal synapses) are created dynamically when learning is enabled. This affects the entire temporal phase of HTM. For simulation, randomized test vectors are created that assume all cells in the network have between zero and eight distal segments (or between zero and one hundred twenty eight distal synapses). This is implying that the network has learned a significant portion (i.e. up to 50% of the total number distal connections could have already been created). How this relates to KTH is unknown, but we are assuming it represents a more saturated network than the KTH inputs would represent, and thus most likely represents a longer run time.

### 4.3.3 Data Precision

IEEE half precision is implemented as to maintain some compatibility with general purpose in the SIMD solution, which offers some advantages. Other algorithms can easily be implemented into the SIMD architecture to run, and possibly even optimize the critical algorithms. Also, half-to-single precision and single-to-half precision conversion can be implemented via the ISA so that the core can easily communicate through memory with other processors. The disadvantages of floating point protocols are larger area, longer delays, and larger power consumption.

Floating point is not needed for the HTM algorithm. It is introduced in temporal phase one in the step to create new distal segments to accelerate execution. In this step, when an active column wants to create a new distal segment on a cell, it needs to connect that distal segment to cells in

16 other active columns. Locality is enforced by prioritizing the nearest active columns first. To implement this, the Euclidian distances from each active column to every other active column is calculated and used to sort the active columns on each node. With the current HTM network baseline (1664 columns), some distances are large enough that half precision cannot accurately represent the value. For this network, at least ten bits for the integer component of the value would be needed in a fixed point solution, with only 6 bits needed for the fractional component.

Execution precision for Sparsey in the SIMD exploration is limited by characteristics of the network. Early work involved investigating if 8-bit floating point was wide enough to accurately implement Sparsey but still provide programmability. 8-bit floating point could only represent values between -31 to positive 31, with fractional precision of $1/8^{th}$. For the Sparsey baseline for SIMD, the largest number needing representation in floating point comes in normalization step for the first layer, when the pixels in the input of a mac are summed. The remainder of Sparsey relies primarily on fractional precision, which 6-8 bits should suffice. In 16-bit half precision, the significant is represented by ten bits, which for values between -10 to 10, enough fractional precision for implementing probabilities and probability mass functions exists.

### 4.3.4 Scalability of Solution

In regards to scalability, SIMD primitive has been designed around a set of assumptions. The first assumption is that data and instructions are accessed directly from scratch pad memories. For this work, these memories have been SRAMs (16-bit for data, 48-bit for instructions). The second assumption is that these memories are single-cycle access and the architecture is currently locked to that assumption. This assumption affects when the decode stage expects a new instruction and when the load-store unit expects the results of load operations.

The third assumption is that the problem is always programmed to fit into the memory allotted to a SIMD primitive, and that there that there are enough memory banks and ports to fulfill the bandwidth required to feed the processing elements at 250 MHz. For the current Sparsey baseline, 311 Mb of distributed memory across the primitive would be needed for storing the weights and state vectors, while only 3 Mb of distributed memory across the primitive is needed for ongoing probability execution. For the current HTM baseline, 1.13 Gb of distributed memory is needed to store the entire state of the network, assuming that memory is pre-allocated for max size of distal segment and distal synapse data variables. Another 29.4 Mb of distributed memory would be needed for storing the ongoing variables between steps of the algorithm. Using 28 nm memories, Sparsey would require ~5 $mm^2$ of SRAM, while HTM would require ~120 $mm^2$ of SRAM. When operating on the SIMD unit, each unit is accessing a single port of the SRAM, which equates to 128 Gbps bandwidth to memory. When PIMs are operating, the worst case memory bandwidth occurs for the weight summation PIMs. There are 16 weight summations PIMs per node, each accessing two ports of the SRAM, which results in a required memory bandwidth of 4 Tbps.

The primitive can be designed to assume a growing problem size for two different scalable paths. The first path is if silicone is not increased beyond the primitive, but the input image size is increased and the algorithm network must grow. This mainly affects the weights in Sparsey, and the network information in HTM. The memory requirements for Sparsey weights are always pre-known based on the network, but HTM is dynamic and depends on the data the network

must learn. The architecture could be updated to support pointer manipulation to better suit HTM data requirements but at large increase to execution latency. Additionally, once memory access logic grows, the memory access latency would increase beyond a single cycle. Accessing DRAM instead of SRAM will only further degrade performance.

The second path is if silicon area is increased. For every doubling of the image input, the algorithm network must be quadrupled in size and another layer added. For both Sparsey and HTM, to decrease primitive-to-primitive communications, each primitive could be considered its own network, and the image input could be overlapped around the edges of each primitive to catch the correlation between pixels across networks. Then one primitive could be programed to take the outputs of all primitives and execute the final layer. This would ensure the max amount of parallelization with the least amount of communication between primitives.


Approved for public release; distribution unlimited.

# 5.0 ASIC IMPLEMENTATION OF SPARSEY

Cortical algorithm, inspired by the neocortex, offers promising breakthrough to build the next generation massively parallel, extremely power efficient and highly scalable systems. Because of the inherent parallelism of these cortical algorithms, a suitable parallel computational platform is required to build system based on cortical algorithms. The use of multi-core central processing units (CPUs) and general-purpose graphic processing units (GPGPUs) is a very promising platform to support the development of large-scale systems based on cortical algorithms. Multi-core processor achieves good performance (speedup) compared to the traditional sequential processor but unit of work per core doesn't scale very well. Besides, CPUs are poorly suited for the large-scale networks. As a result, cortical algorithm doesn't provide good performance on multi-core platform. On the other hand, a modern GPGPU provides hundreds of streaming cores and thousands of threads, which is an attractive choice of parallel computing platform for large-scale network. GPGPU performs poorly when certain-class of cortical algorithms is mapped to GPUs, which requires to move huge amount of data (i.e., weight matrices) from DRAM to local memory before start parallel computation. Other than the data transfer overhead, hotspot function (i.e., weight summation of each cell) impacts the performance. These technological issues can be resolved using ASIC with dedicated local storage for weight matrices (i.e., increase memory bandwidth), custom units to handle hotspot and exploiting other approximations (i.e., fixed precision floating points, quashing function etc.). So there exists a greater need to explore the custom hardware solution, especially exploiting the behavior of cortical algorithms. In this section, a purely configurable custom hardware solution of Sparsey$^{TM}$ having massive speedup over GPU baseline is proposed.

The primary goal of configurable ASIC is to build hardware-accelerated platform of Sparsey$^{TM}$, which will achieve better speedup over the GPU baseline. Besides that, the custom design leverages to achieve better performance of other matric (i.e., area, power etc.) by exploiting the behavior of Sparsey$^{TM}$.

This is a design as well as an analysis of multi-layers Sparsey consists of configurable independent Mac processor ASIC for arbitrary sparse input feature vectors, implemented with 65 nm complementary metal-oxide semiconductor (CMOS) technology. The Mac or PE processor is composed of a group of CMs and a CM module consisting of arbitrary number of cells with storage for weight matrix. Compared with CPU/GPU baseline, this multi-layer architecture based on ASIC Mac processor has enhanced speed, reduced power as well as hardware complexity as it utilizes dedicated storage for synaptic storage, deeper pipeline, shared arithmetic processing unit, and shared registers. Then, the three layers baseline model was analyzed with configurable ASIC Macs. Finally, being synthesized and place-and-routed with 65nm CMOS technology, it gives ~145X performance speedup, ~20X power reduction and overall ~2644X performance/power improvement. Even at 65 nm, it also gives a ~4X area reduction compared the die area of 28 nm Tesla K20C GPU at 561 mm$^2$.

This section describes a configurable ASIC implementation of Sparsey. The first subsection describes the architecture and design of a cluster and intra-cluster interconnection topology. The second subsection presents the results of custom hardware solution and comparison with GPU


Approved for public release; distribution unlimited.

baseline. The final subsection discusses the mapping of Sparsey into the ASIC, data precision and scalability solution.

## 5.1 Architecture and Design

### 5.1.1 Cell

Cell or neuron is the basic computational unit of Sparsey architecture. A single cell can have two synaptic input matrices: a bottom-up (*U*) and a horizontal (*H*). The hardware architecture of a cell is shown in Figure 8. The submodule "Weight Sum" computes the summation of synaptic weights for both *U* and *H* inputs coming from SRAM. This is the major hotspot of Sparsey and to eliminate the hotspot, a pipeline weight sum submodule was designed in a way that it takes 256 bits input (2R ports of SRAM and 128-bit wide) and do pipeline addition. The area and average power of single "Weight Sum" submodule is 6904 $um^2$ and consumes 5.1 mW when using 65 nm technology running at 170 MHz clock frequency. The hardware architecture of "Weight Sum" submodule is shown in Figure 7. A single cell performs algorithm steps 1 (i.e., hotspot), Step-2, Step-3, and Step-7 of Table I.B-1 of Technical Report 1.

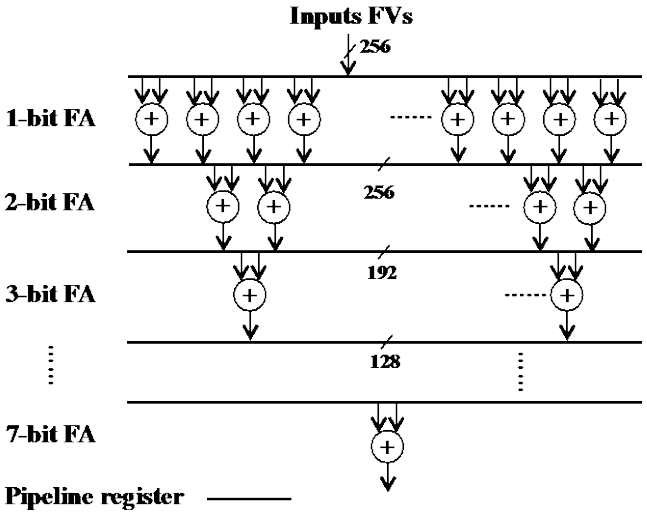


**Figure 7: Hardware Architecture of "Weight Sum" Submodule**

### 5.1.2 CM

A minicolumn or CM module consisting of 16 physical cells, winner take all (WTA), probability distribution submodule and shared memory. The probability distribution submodule calculates the probability of each cell inside CM and this submodule is just a bunch of 16-bits addition and division logic. On the other hand, the WTA module performs Softmax i.e., select a final winner cell in each CM according to the ρ distribution. The binary search has been used to find out the winner from a sorted ρ distribution array. A shared memory of size 0.25Kb is used to store the intermediate values i.e., weight summation, normalized weight summation, probability and cumulative probability of each cell.


Approved for public release; distribution unlimited.

### 5.1.3 Mac/PE

Macrocolumn (Mac) or processing element (PE) is a group of 8 CMs. The hardware architecture of a single PE is shown in Figure 8. Each PE can be configured up to 32 CMs based on application area. It also has one “Weight Sum”, one arithmetic logic unit, and shared memory having size of 2.25 Kb.

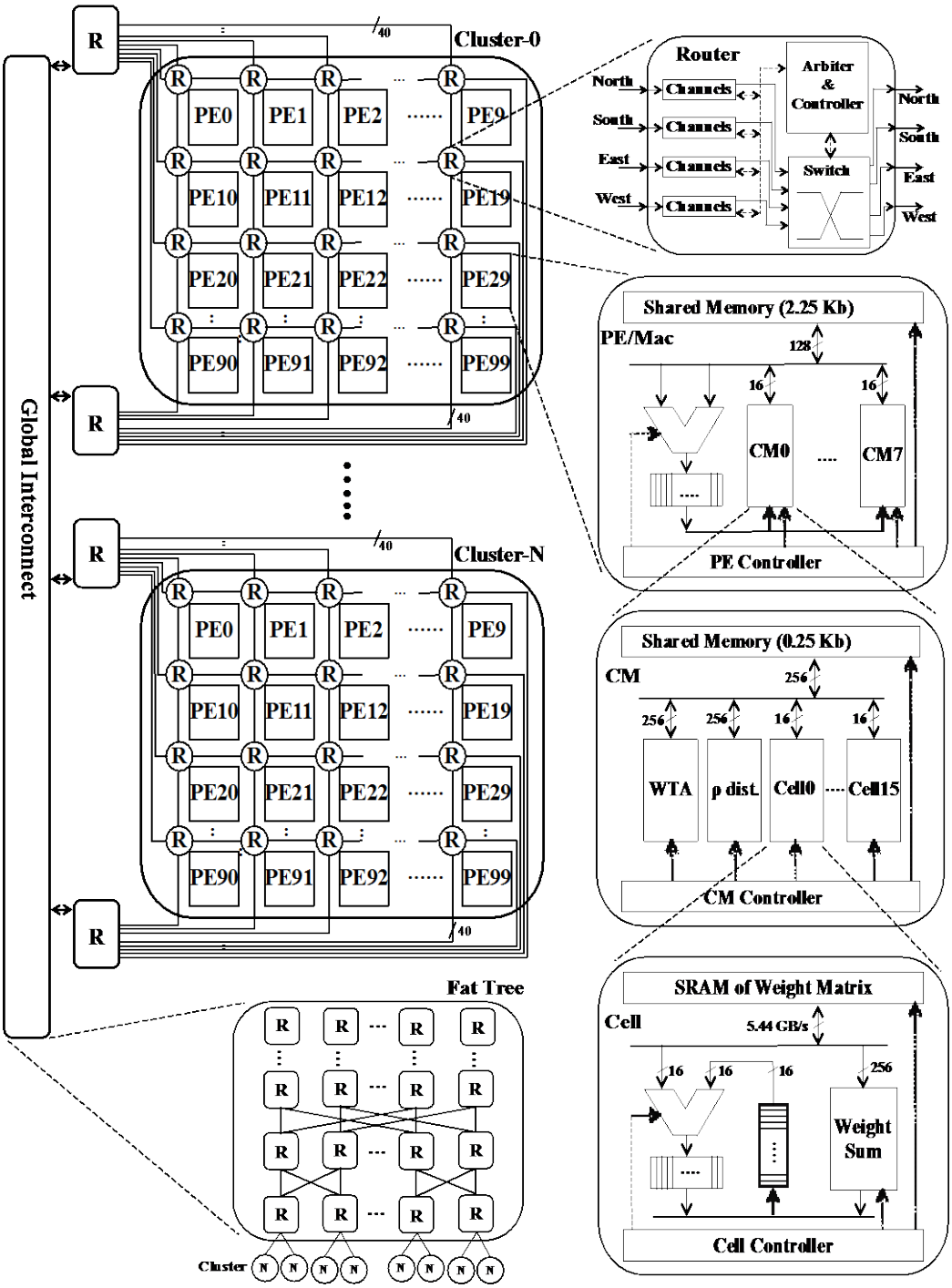


**Figure 8: Overall ASIC Architecture of Sparsey**


Approved for public release; distribution unlimited.

The submodule “Weight Sum” computes the summation of input feature vectors. To achieve better performance, the same “Weight Sum” submodule of cell was used. The arithmetic logic unit computes the global familiarity (G) and expansivity ( $\eta$ ) of quashing function aka algorithm step-5 and step-6. The shared memory was used to store the sparse output of each PE after training/recognition phase of each frame. Each PE is a parameterized design of Sparsey. The existing 16 physical cells/CM and 8 physical CMs/PE can be configured based on application areas. The configurable parameter of a single PE is given in Table 8.

**Table 8. The Configurable Units of Single PE**

| Name | Configurable Units | |
|---|---|---|
| | **Minimum Value** | **Maximum Value** |
| Cells/CM | 8 | 64 |
| CMs/PE | 4 | 32 |
| Input bits/Cell | 128 | 512 |
| Expansivity ( $\eta$ ) | 0.01 | 0.99 |
| Total Cells | 32 | 2048 |

### 5.1.4 Cluster

A cluster is a 2D array of PEs or Macs. The basic operation that takes place in each step is that every PE in cluster computes the sparse outputs after training/recognition of each frame and store output to its shared memory. PEs inside a cluster communicates through intra-cluster interconnection. A cluster can have arbitrary number of PEs. In order to explore the performance of custom ASIC with the Sparsey GPU baseline (Table 4.2.4) we have considered 10x10 PEs in a single cluster. The core area (i.e., without storage of weight matrices) of a cluster is 220 $mm^2$ using 65 nm technology. During scalability analysis in discussion section, we have considered multiple clusters connected through global interconnection network.

### 5.1.5 Intra-cluster NoC

Sparsey has physical locality during inter-PE communication i.e., each PE communicates only its nearest neighbor. Based on the inter-PE communication, the *mesh* on-chip network-on-chip (NoC) is chosen for intra-cluster NoC. Moreover, mesh network can be operated at high speed without repeaters and latency is much lower. To do simulation of baseline model (Table 14), the C++ interconnection simulator [mesh] was configured to the baseline model. The traffic generator module of the simulator is changed in way so that it generates packet like RTL PE. This cycle accurate simulator gives the number of cycles for inter-PE communication within a cluster. The configuration of interconnection network is given in Table 9.

### 5.1.6 Global Interconnection

The global interconnection network connects the global memory and supports inter-cluster communication. During analysis of the baseline model, a single cluster has been considered. So the role of global interconnection is negligible. But the role of global interconnect network is significant on scalable architecture which was included in discussion section. The *fat tree* topology is chosen as global interconnection. The reasons of using fat tree as global interconnect

network is that it offers path diversity which holds fault tolerance property. Besides that it scales better for hierarchical network, also so does with respect to cost [fattree]. Moreover, it has very less impact on performance because of static routing. A cycle accurate simulator *booksim* was taken from [booksim] for global interconnection latencies during scalability studies in discussion section.

**Table 9. Intra-cluster NoC Properties**

| Name | Value |
|---|---|
| Channel per port | 1 |
| Channel width | 40 bits |
| Flit size | 32 bits |
| Flits/packet | 25 |
| Channel buffer size | 4 flits |
| Delay (Arbiter + Controller) | 3 |
| Routing delay | 1 |
| Routing scheme | XY |

**Table 10. Inter-cluster NoC Properties**

| Name | Value |
|---|---|
| Topology | Fat tree |
| Number of switches | 48 |
| Number of channels | 256 |
| Flit size | 32 bits |
| Flits | 1 |
| Maximum number of clusters | 64 |
| Channel buffer size | 64 lits |

## 5.2 Results

**Table 11: The Number of Cycles of Weight Summation of the Baseline Model**

| Layer | Input Type | # of Cycles | Total Cycles | Time (µs) @170 MHz |
|---|---|---|---|---|
| Layer-1 | Bottom-up | 27 | 522 | 3.13 |
| | Horizontal | 495 | | |
| Layer-2 | Bottom-up | 774 | 1071 | 6.43 |
| | Horizontal | 297 | | |
| Layer-3 | Bottom-up | 180 | 279 | 1.67 |
| | Horizontal | 99 | | |
| **Total** | | | **1872** | **11.23** |


Approved for public release; distribution unlimited.

**Table 12. The Area and Power Report of Submodule**

| Name | PE | CM | Cell |
|---|---|---|---|
| Technology | 65 nm | | |
| Speed | 170 MHz | | |
| Logic area ($mm^2$) | 2.170 | 0.270 | 0.016 |
| Average Power (W) | 0.880 | 0.110 | 0.006 |
| # Of Gates (K) | 1507 | 188 | 11 |
| # Of DFF[1] (K) | 129 | 16 | 1 |

[1]The number of DFF includes the share memory

**Table 13. On-chip NoC Latencies for Different Layers of Sparsey**

| Layer | Operation Name | Cycles | Time (µs) @250 MHz |
|---|---|---|---|
| Layer-1 | Receive horizontal inputs | 236 | 0.94 |
| Layer-2 | Receive bottom-up inputs | 1236 | 4.94 |
| Layer-2 | Receive horizontal inputs | 126 | 0.50 |
| Layer-3 | Receive bottom-up inputs | 136 | 0.54 |
| Total | | 1734 | 6.92 |

**Table 14. The GPU Baseline Model and Processing Time of GPU and ASIC**

| Sparsey™ Baseline | | | | | Processing Time | | Acceleration |
|---|---|---|---|---|---|---|---|
| Layer | Mac/Layer | CM/Mac | Cell/CM | Total Cells | GPU (µs) | ASIC (µs) | |
| Layer-1 | 100 (10 x 10) | 20 | 40 | 80000 | 5730 | 15.01 | 381.75 |
| Layer-2 | 4 (2 x 2) | 20 | 40 | 3200 | 1640 | 21.06 | 77.87 |
| Layer-3 | 1 (1 x 1) | 20 | 40 | 800 | 490 | 11.56 | 42.39 |
| NoC | | | | | | 6.92 | |
| **Total Processing Time and acceleration** | | | | | 7938 | 54.55 | **145** |

* The PE is synthesize and place-route for ~6ns clock

**Table 15. Speed/Power Comparison between ASIC and GPU**

| Factor | GPU Baseline 28 nm | ASIC 32 nm | ASIC 65 nm |
|---|---|---|---|
| Speed | 8 ms | 0.055 ms | 0.055 ms |
| Core area[1] | 561 $mm^2$ | 122 $mm^2$ | 220 $mm^2$ |
| Average power[2] | 200 W | 11 W | 22 W |
| Speed/Power | 1 | 2644 | 1322 |


Approved for public release; distribution unlimited.

## 5.3 Discussion

### 5.3.1 Mapping Sparsey to the ASIC Architecture

Sparsey has two modes of operation: i) learning, and ii) recognition. During the learning mode, each cell within CM performs six steps; each CM within Mac/PE performs two steps; and each PE/Mac within cluster performs two steps. After each learning phase, one cell from each CM is selected based on its probability. The six steps of each cell are: summation of weight matrices (i.e., memory read and addition logic), normalizing the weight summation (i.e., division logic), computing match (i.e., multiplication logic), generating an activation function or quashing function (i.e., division, multiplication, and addition logic), calculating probability (i.e., division logic), and finally updating the memory (i.e., only winner cell of each CM). On the other hand, single CM finds out the local likelihood (i.e., find out maximum value), and selects a winner cell (i.e., search operation). The PE calculates the global familiarity (i.e., addition and division operation), and slope of transfer function (i.e., addition, multiplication, and division operations). In recognition phase, each cell performs first three steps and then, cell within CM having highest weight summation will be selected as maximum likelihood estimation. There will not be any memory update during recall/recognition phase.

The first operation of each cell i.e., memory read and addition of weight matrices was the hotspot of Sparsey in GPU baseline. It consumes almost 80% processing time of GPU implementation because of limitation of memory bandwidth. After the hotspot detection, it was clear that the weight matrix of each cell is local in nature and can have a lot of advantages from local high bandwidth memory for each cell. In order to eliminate the effect of hotspot, a dedicated 2-port SRAM (128 wide) was assigned per cell. The peak memory bandwidth needed for the SRAM is 5.44 GB/s. The energy consumed for each SRAM read access is 44.16 pJ as per 65 nm memory generator.

Moreover, to do computation at a much faster rate and increase the throughput a pipeline submodule (Figure 7) was designed to sum the weight matrix. The area and power of weight sum submodule was obtained to be 6904 $\mu m^2$ and 5.1 mW, respectively using 65nm technology operating at a 170 MHz clock frequency.

The second step of each CM is to select a winner cell based on probabilities of cells in that CM and a random number. This is also called winner-take-all step. The binary search algorithm is implemented to select a winner cell from the cumulative probabilities of all cells in a CM. The area of average power of single binary search submodule is 3654 $\mu m^2$ and 0.574 mW, respectively by using 65 nm technology for 170 MHz clock frequency.

The mesh on-chip NoC was chosen for intra-cluster communication. The latency of on-chip NoC is 6.92 μs of three layers Sparsey baseline for 250 MHz clock frequency.

### 5.3.2 Reduced Precision Floating Point

Sparsey is not a floating-point intensive cortical algorithm. However, some floating points is used but the accuracy of the algorithm doesn't depend upon the precision of floating point implementation. Instead, it depends upon the behavior of random numbers and the input patterns. In the ASIC implementation, we used two-to-four digits of precision for floating numbers.

In order to implement the fixed precision (i.e., 2-digits and in some steps 4-digits) floating point, the *dividend* was multiplied by a decimal number (i.e., 1e2 or 1e4) and then integer *divider* was used to get better performance. The resorting division algorithm was used to design as well as implement the division logic. The complexity of restoring division algorithm is O(n) where n is the number of bits in *dividend*. The area and power of single 32-bit divider is 1252 $\mu m^2$ and 0.9 mW using 65nm technology.

### 5.3.3 Scalability of Solution

A single cluster was sufficient to analyze the GPU baseline model of Sparsey. When the application area or problem size increases, it's not sufficient to solve the problem using a single cluster and this phenomenon motivates us to analyze and to propose scalable custom hardware architecture of Sparsey. The scalability of ASIC architecture falls into two categories: i) without increase of silicon ii) with increase of silicon.

**i) Without increase of the silicon area:** Single cluster has 10x10 PEs consisting of 12800 physical cells. The total memory required of a cluster is 800 Mb with peak bandwidth of ~500 Tbs (i.e., ~5 Tbs/PE) and the required processing time is ~0.05 millisecond for single frame. The area and power of a single cluster is 220 $mm^2$ and 20 W using 65 nm technology for 170 MHz clock frequency. For every doubling of the problem size (i.e., input feature vector count) an extra layer would be needed, which requires a single cluster to be refilled with 800 Mb weight matrices within 0.05 millisecond (i.e., processing time) from global memory. It requires huge amount of memory bandwidth (~16 Tbs) between global memory and on-chip memory. One way to decrease the global memory bandwidth requirement is to increase the on-chip memory size. For every doubling of the on-chip memory size, the global bandwidth requirement will be decreased by factor of 2. The speed, power, and interconnection latencies of Sparsey network using single cluster is shown in Figure 9 considering enough on-chip memory.


Approved for public release; distribution unlimited.

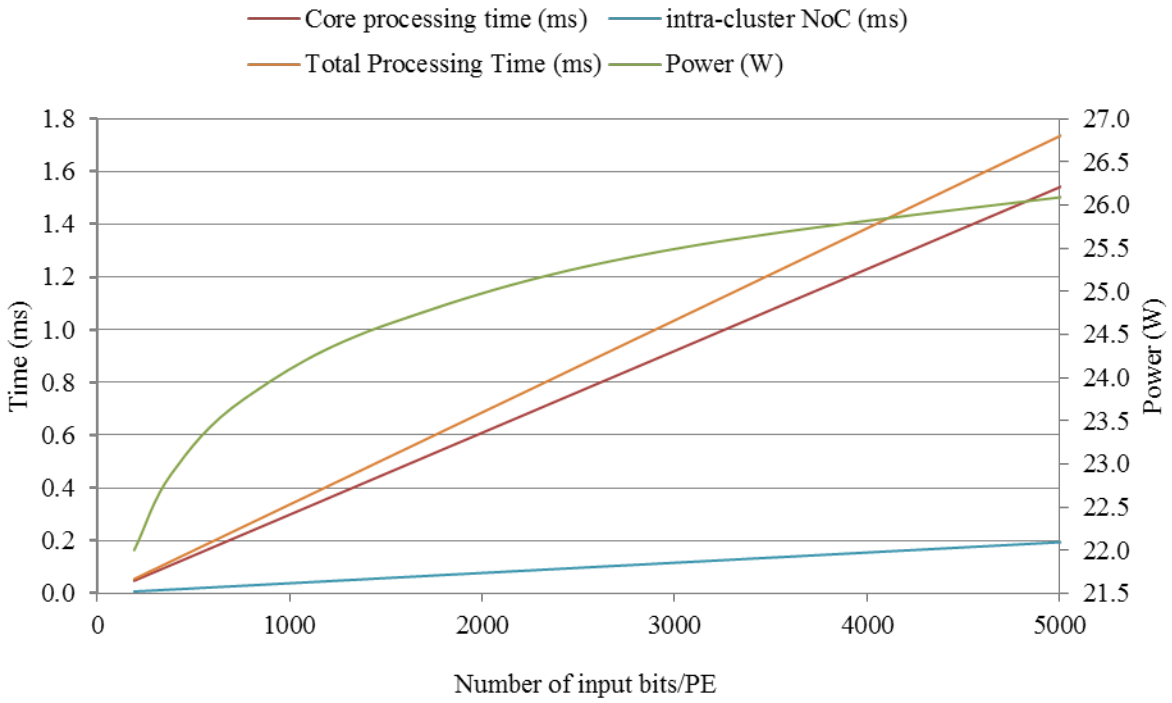


**Figure 9: Scalability Analysis without Increase of Silicon (i.e., core area 220 mm$^2$) for 65 nm Technology and 170 MHz Clock Frequency**

**ii) With increase of the silicon are:** In this case, silicon area increases twice for every doubling of input feature vectors and also adds an extra layer. The inter-cluster communication through global interconnection is required for normal cluster operation. There could be two possible scenarios: i) each cluster may consider independent network and edge-PEs of cluster doesn't require to communicate with PEs of adjacent cluster; ii) edge-PEs of adjacent cluster needs to communicate with each other. For the sake of simplicity, we have considered each cluster as an independent network and to be responsible for specific input feature vectors. Since the top-level network needs data from bottom-layer clusters, the Fat tree global interconnection network is chosen for inter-cluster communication. The speed, power, and interconnection (i.e., local and global) latencies of Sparsey network is shown in Figures 10, 11, and 12.


Approved for public release; distribution unlimited.

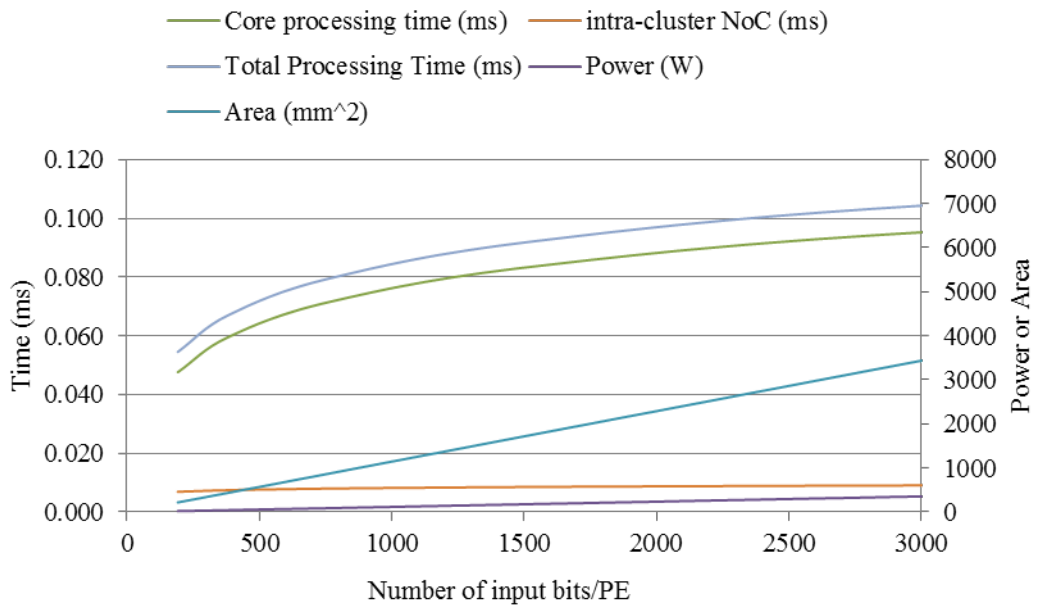


**Figure 10: Scalability Analysis with Increase of Silicon for 65 nm Technology and 170 MHz Clock Frequency**

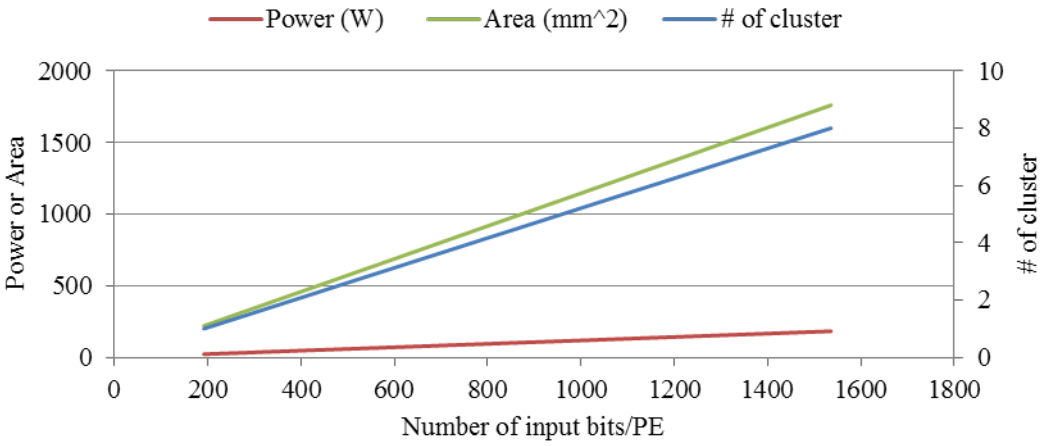


**Figure 11: Scalability Analysis of Power and Area with Increase of Silicon (i.e., clusters) for 65 nm Technology and 170 MHz Clock Frequency**


Approved for public release; distribution unlimited.

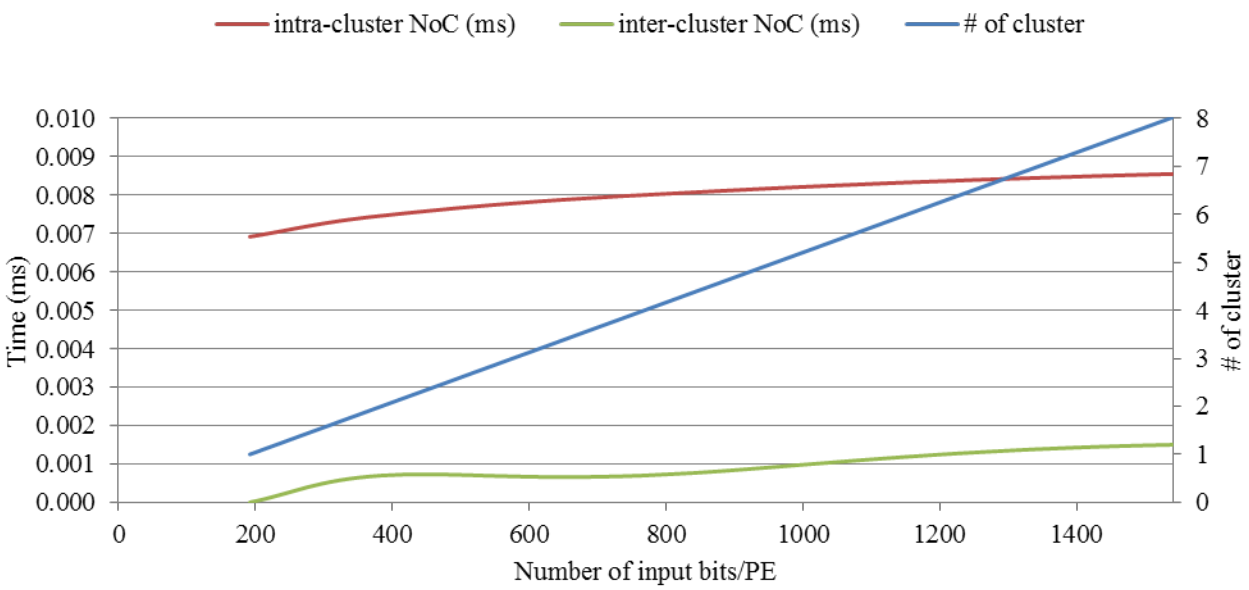


**Figure 12: Scalability Analysis of Interconnect Network with Increase of Silicon (i.e., clusters) for 65 nm Technology and 170 MHz Clock Frequency**


Approved for public release; distribution unlimited.

# 6.0 ASIC IMPLEMENTATION OF HTM

## 6.1 Architecture and Design

In this part of the report, a configurable custom ASIC design for the HTM algorithm is proposed. The primitive unit in this proposed design is called the processor core. Each processor core is designed to provide the computation capability required by a typical neural network with reasonable performance. Based on the data from Numenta, a typical neural network has 2048 columns and 32 cells within each column, which is the baseline network in our GPU implementation.

The schematic of proposed processor core is shown in Figure 13. Each processor core consists of multiple identical processing elements and one central processor. During the processing progress, the target neural network is divided into several sub-regions and each processing element is responsible for the computation of one region. Within the processor core, all the processing elements are connected to the central processor, so that the data communication among processing elements can be replaced by the data communication between central processor and processing elements. In addition, the central processor is also responsible for the inter-core communication and the computation which may require data from all elements such as inhibition, to reduce the duplicate data transaction among processing elements.

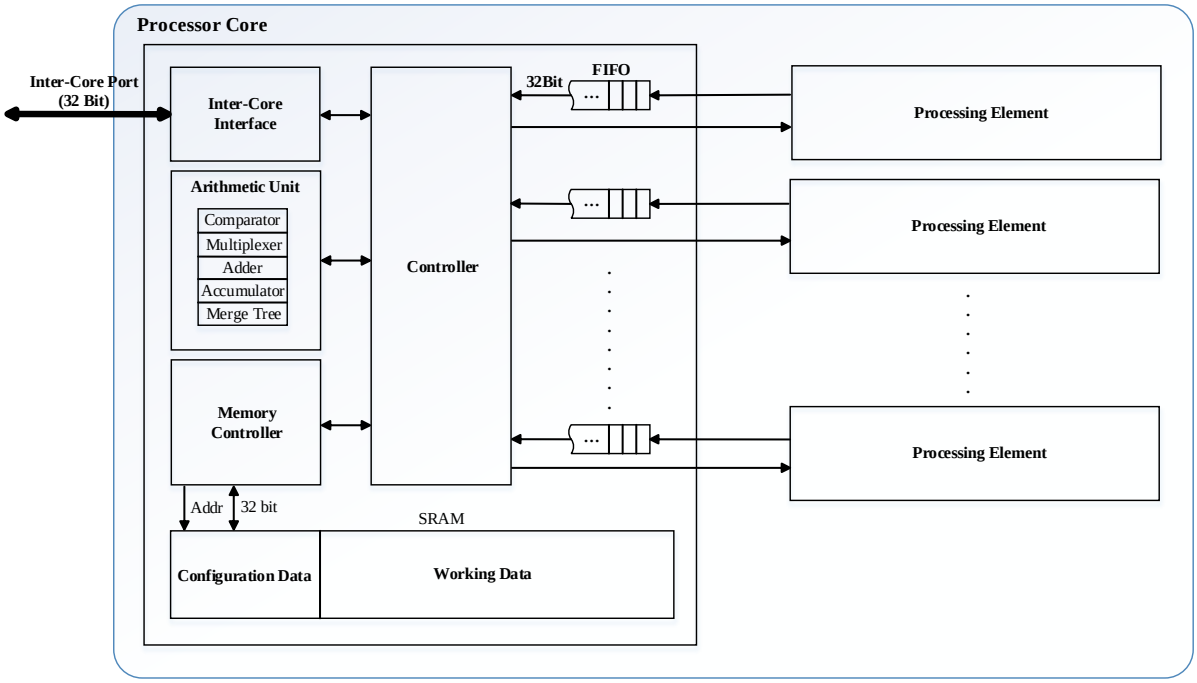


**Figure 13: Schematic of Processor Core**

The schematic of the proposed processing element is shown in Figure 14. The processing element consists of multiple identical execution lanes, element control logic and interface logic. In addition to the basic arithmetic operation such as add, multiplication and accumulation, the execution lane can also provide special operations required by HTM, such as data matching, array sorting and array merging. Dedicated memory device which is SRAM in current version is


Approved for public release; distribution unlimited.

assigned to each execution lane in each processing element to store the biological information of neuron network, such as the proximal synapse, distal synapse and synapse permanence. The dedicated memory device is only accessed by the corresponding execution lane, so that there will be no memory access confliction among execution lanes.

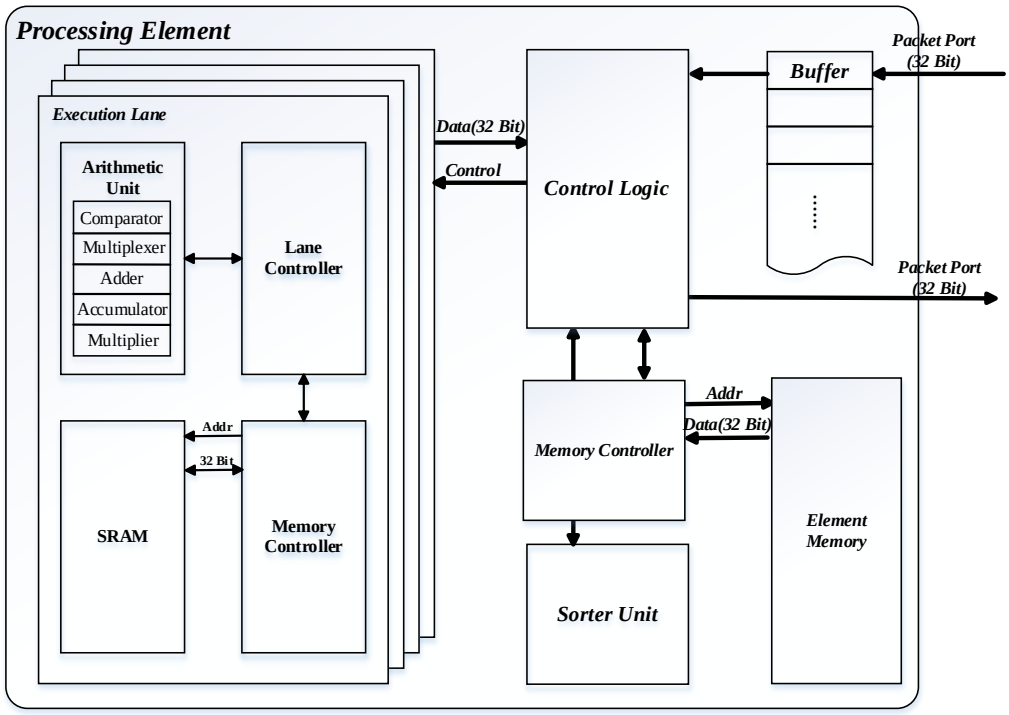


**Figure 14: Schematic of Processing Element**

In order to implement larger networks, multiple identical processor cores on the same silicon chip can be connected together as the ring network shown in Figure 15. The inter-core data communication can be divided into three steps: synchronization, data sending and data forwarding. Though all the processor cores within the network are triggered simultaneously and share the same algorithm, processor cores can be in different processing stages due to the dynamic distribution of workload in practice. As a result, processor cores reach the communication stage earlier have to be stalled until all the cores done. In the first step, each processor core is designed to send a "Done" message to all the other nodes in the network after certain processing stage and then come into the waiting state. In the meantime, the counter within the interface unit of each processor core is accumulated for each received "Done" message. The received "Done" message is also forwarded to the next node. In the second step, the processor core which has received "Done" message from all the other nodes within the network starts to send the local data to the neighbor cores and saves the data from other cores into local memory until all the local data is sent. In the third step, the processor core is designed to forward data received from other nodes until all the data have been forwarded. In the proposed design, there is one input channel and one output channel. The word width of each channel is 32 bit and the maximum inter-core bandwidth in proposed design is 3.2 Gbs. For the data communication among ring networks on various silicon chips, a router is added into the network due to the influence of transmission latency, signal integrity and timing. In that case, the


Approved for public release; distribution unlimited.

entire procedure is divided into two steps, inter-core transaction and inter-ring transaction. The inter-ring transaction shares the similar procedure as the inter-core one. The router in each ring is responsible to receive the data from the other rings and then broadcast it to all the processor cores within the network.

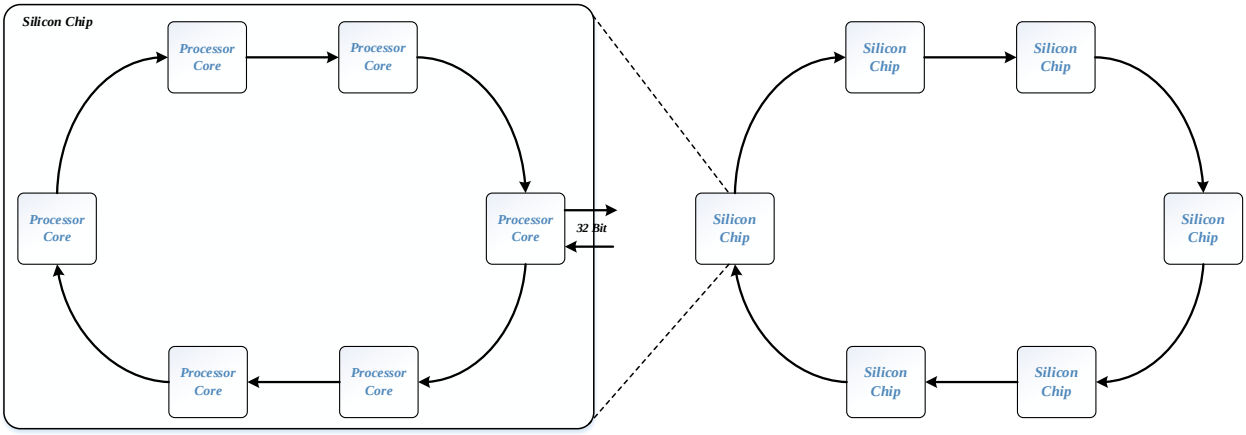


**Figure 15: Schematic of Inter-Core Network**

## 6.2 Results

**Table 16. Baseline Network Size**

| Network Features | Value | Total Distal Synapse | Total Proximal Synapse |
|---|---|---|---|
| Column Matrix Dimension | 38 * 52 | 11308170 | 79040 |
| Proximal Synapse Per Column | 40 | | |
| Neuron ( Cell ) Per Column | 32 | | |
| Distal Segment Per Cell | 15 | | |
| Distal Synapse Per Segment | 12 | | |

**Table 17. Memory Requirement of Proposed ASIC**

| Proximal Synapse | | Memory Per Lane (Mb) | Word Size | |
|---|---|---|---|---|
| Total Memory (Mb) | 1.6 | 8.2 | Synapse Info | Permanence |
| **Distal Synapse** | | | 32 Bit | 16 Bit |
| Total Memory (Mb) | 522 | | | |

**Table 18. Performance Comparison between GPU and ASIC**

| Subject | GPU | AISC | Improvement |
|---|---|---|---|
| Processing Time | 225.7 ms | 0.63 ms | 358x |
| Core Area | 561mm$^2$ | 477.8mm$^2$ | 1.2x |
| Average Power | 70 W | 298 mW | 235x |

*The clock period of ASIC is 100 MHz@65nm


Approved for public release; distribution unlimited.

## 6.3 Discussion

### 6.3.1 Hierarchical Processor Core and Network Topology

Though most of the operations in HTM are independent within each neuron column or cell, individual column or cell has to require data such as overlap value and segment activity from all the other columns or cells within the same region in certain operation. In the architecture discussed herein, this operation is implemented by broadcasting data stored in each processing element to all the others, which can resulted in numerous data movement and duplicate data processing in each node. For instance, the sorting operation for overlap value from the entire neural network when in inhibition has to be separately performed within each processing element. In the proposed design, by merging the sorted overlap value from each element in parallel in the central processor as shown in Figure 16, each processing element is only required to sort and send the overlap value stored locally and the sorting in element and merging in core can be implemented in pipeline to further reduce the operation latency. As a result, the total number of data movement among processing elements is significantly reduced with an improved performance.

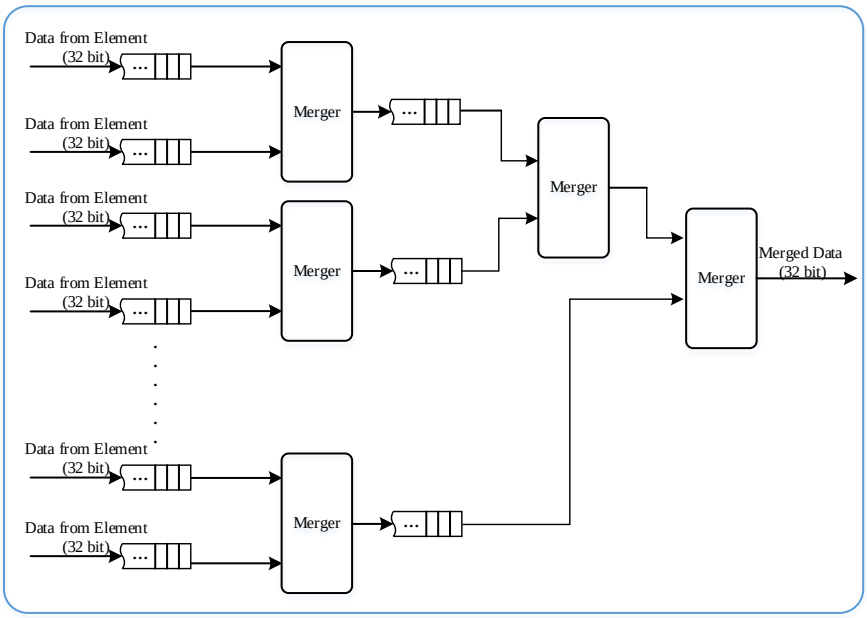


**Figure 16: Schematic of Merging Tree in Central Processor**

In addition to the performance improvement, the hierarchical architecture can provide an efficient solution to the synchronization problem among processing elements which is resulted from unbalanced workload. Though the same algorithm is shared among all processing elements, the practice progress in each individual element can be different due to the dynamic distribution of active columns. To maintain the correct timing among elements, each processing element is required to send a “Done” message to the central processor after completing the operations with dynamic workload, such as prediction operation and learning operation. Meanwhile, the processing element is not allowed to move forward to the next operation until the “Done” messages from all the elements have been collected in the central processor.


Approved for public release; distribution unlimited.

Considering the operations in spatial pooling and temporal pooling, the proposed network can provide the following benefits: 1) Data stored in any node (process core) can be sent to all the other nodes within the same network; 2) The communication latency should be no longer than the 10% of the total processing time; and 3) Nodes in the network can send the local data simultaneously since transaction is not triggered until local data in all the processor cores is ready. For each input, only a small portion of columns (2%) mapped in each processor core are required to send data such as column index, cell index and overlap value to other processor cores in HTM and the detail of data packet is shown in Table 19.

**Table 19. Inter-Core Communication Packet**

| Inter-Core Packet Content | Packet Width | Packet Count Per Input |
|---|---|---|
| Max Overlap Cycle | 16 | 1 |
| Max Active Cycle | 16 | 1 |
| Overlap Value | 32 | n |
| Column Index | 32 | n |
| Active/Learn Column Index | 32 | n |

n is the total number of active column

Based on the result from C++ simulator, the proposed ring network with totally 8 baseline processor cores requires about 6.2us to complete the data transaction for each input, which is only 1% of current baseline processing time. Compared to other qualified topology candidates with better transaction latency such as mesh, bufferfly or crossbar, the proposed ring network requires fewer port number, smaller buffer size and simpler logic to implement as well.

### 6.3.2 Network Mapping in ASIC Implementation

As mentioned in the previous section, the neural network is divided into multiple sub-regions and the computation workload of each sub-region is mapped into one processing element. Due to the operation synchronization requirement among processing elements, the elements with less workload can't move forward to the next operation until the elements with most heavy workload is done. In the previous ASIC implementation, the neuron network is vertically divided in both spatial and temporal pooling, which means the workload of all columns within the given sub-region and cells within each of these columns is mapped into the same corresponding processing element. From the simulation result, it can be found that some certain columns are selected as the active column more often than the others, though the spatial pooling does pretty well work in distributing active columns across the entire network. As a result, there are more segments created in these more active columns during the temporal pooling and larger number of distal segment can result in longer operation latency in almost all the operations in the corresponding processing elements while other element stay in waiting.

In the proposed design, the neural network is vertically and horizontally divided in spatial pooling and temporal pooling respectively. Take the baseline implementation as an example, the 52 * 38 network has been divided into 8 regions in each of which there are 35 columns. Instead of mapping the 32 cells in each of these 35 columns into the same corresponding processing elements, neuron cells within column are horizontally divided into 8 layers with 4 cells in each

layer and each processing element is responsible for the workload of one cell layer of entire network as shown in Figure 10. As a result, temporal pooling operations can be performed on different cells of a given active column simultaneously, regardless the column mapping in spatial pooling. The proposed mapping method can significantly reduce the workload difference among processing elements, especially for operations only performed on active columns such as the hotpot in GPU baseline implementation named “CreateSegment”. In addition, by dynamically assigning cell index for each column, the increasing of total distal segment count in each processing element is more balanced during the entire learning progress.

### 6.3.3 Network Scalability

In the HTM, there are two kinds of scalability, spatial scalability and temporal scalability. Spatial scalability refers to the increasing dimension of input data with the size of training set fixed. In the case of that, the neural network requires more columns to ensure that all the input regions can be covered by the proximal synapse while the proximal synapse number in each column stays reasonable. It’s necessary to notice that the number of active columns in each region of the network should be increased as well since a constant density (2%) is desired. However, since the size of training set is fixed, the neuron cell number and maximum distal segment number in each column doesn’t have to be increased. In contrast, temporal scalability refers the increasing size of training set while the dimension of each training data stays same. In the case of that, the neuron network requires more cells in each column or more distal segments in each cell to remember the additional training data. Otherwise, the previous learned data will be overwritten by the new information. The total column number in temporal scalability can stay the same. In practice, the spatial scalability and temporal scalability can happen to the same network simultaneously.

For a given primitive (processor core) in the proposed design, doubling the network size in one region doubles the number of columns mapped into each processing element, which can result in a 100% increase of the spatial pooling operation latency. The performance influence of spatial scalability on temporal pooling is analyzed based on the assumption that the active columns are always evenly distributed across the entire region, so that the average number of distal segment in each column stays almost same as that before scaling. Since the nature of temporal pooling in HTM is actually to check the activity of each created segment existing within each neuron cell, doubling the network size linearly doubles the total distal segment count required to be checked. In addition, due to the increased number of active columns in current region, the latency for processing one segment is increased each operation of temporal pooling. For instance, the latency of matching operation for one segment is increase from 0.74us to 1.22us with a 100MHz clock period. As a result, doubling the network size mapped into one processor core can result in a performance deduction more than 50%. Take the baseline implementation as an example, the average processing time is increase from 0.63ms to 1.34ms while the network size is doubled and the performance deduction is further deteriorated as the size of training set increased. In theory, the number of processor cores should be quadrupled or tripled for every doubling the network size within one region to cover the overhead of inter-core commutation and segment processing in practice. In addition to adding more columns into neuron region, the large network can be divided into multiple regions, so that the total active column number in each region is reduced due to the downsizing and none data transaction is required among regions. As a result, the average processing time is improved for a given number of processor cores. However, another layer should be added to collect the output from different regions in lower layer and it

should be mapped into one or multiple additional processor cores. The memory space required for each additional neuron column is 271Kb.

During the learning progress, the new input is learned by creating and storing a bunch of distal segments in certain neuron cell, so that it can be said that the learning capability of a given network significantly depends on the memory capability in proposed implementation. For each completely "strange" input data in the baseline implementation, about 13.5Kb memory space is required to store the new distal segment. Additionally, the average process time for each input is also increased due to the longer temporal pooling operation latency resulted from the increasing segment number. The average processing time of the proposed baseline implementation versus various sizes of training data is shown in Figure 17. It needs to be noticed that the average processing time will stays almost stable when the segment number in one of the processing element reaches the maximum number.

To deal with performance deduction resulted from the temporal scalability, the number of processor cores should be doubled every time the training size is doubled, so that the average number of distal segment in each column can stay similar. However, the processor core would suffer additional latency once the memory access latency beyond a single cycle due to the growing of memory access logic. Considering the high area cost of SRAM which is the major storage device in our baseline implementation, the 3D-DRAM may be employed in the future. In that case, the DRAM would be divided into multiple regions and each region of them is used as a dedicated memory for the corresponding processor core. The on-chip SRAM memory space will be divided into two regions, the work region and the buffer region as shown in Figure 18, so that the processing logic doesn't have to be stalled to wait for the data being pulled from DRAM.

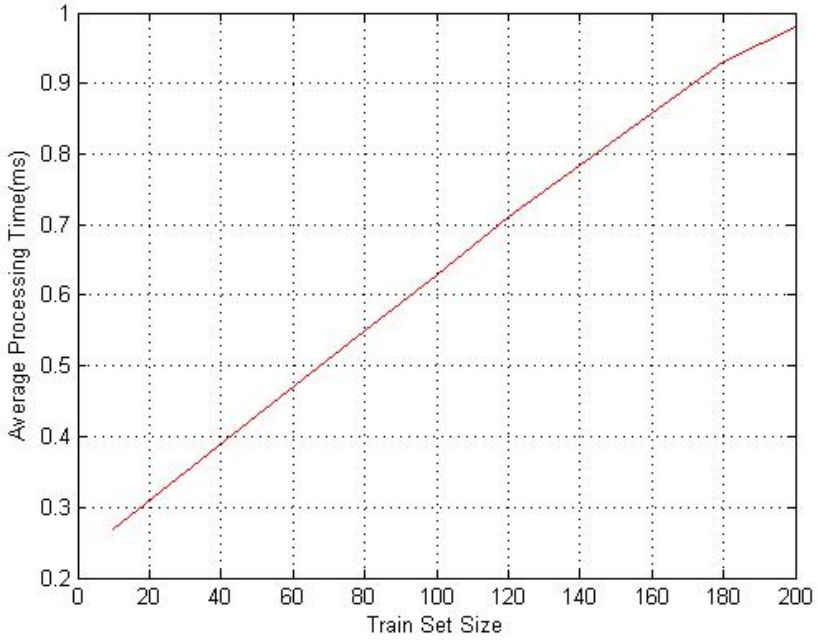


**Figure 17: Average Processing Time vs. Training Set Size**


Approved for public release; distribution unlimited.

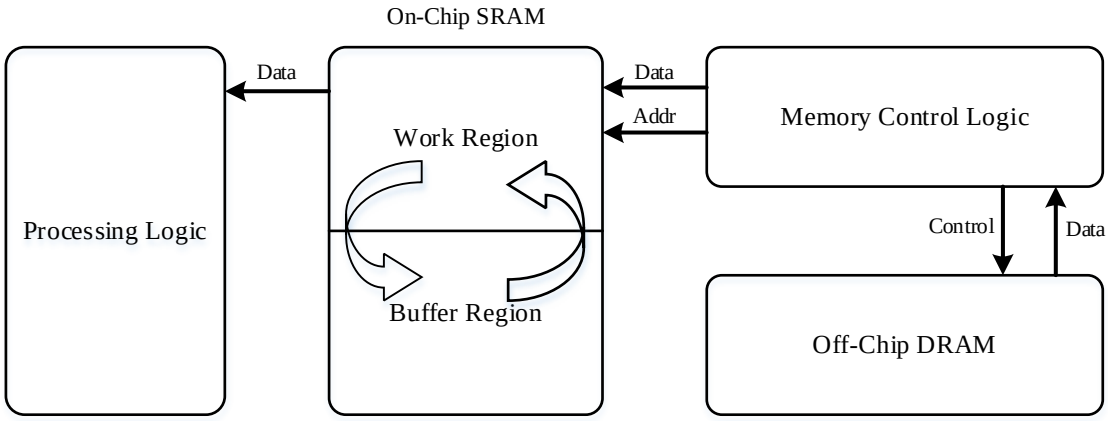


**Figure 18: Dataflow between Off-Chip DRAM and On-Chip SRAM**


Approved for public release; distribution unlimited.

# 7.0 POTENTIAL IMPLEMENTATION USING 3DIC TECHNOLOGIES

## 7.1 Overview

The memory required to store various families of Neural Networks is usually large, often amounting to hundreds of megabytes and perhaps even gigabytes. In addition, many applications require real-time performance meaning a neural network must perform its computations in a few tens of milliseconds or less. Add to this that many system solutions that employ neural networks may not employ only one but many neural networks all providing information to a central system. Considering neural network weights, inputs etc. are stored in memory, all this leads to extremely high memory bandwidth requirements. The raw bandwidth can be achieved using SRAM but SRAM does not realistically provide the required capacity to scale. The capacity can, for the most part be met by DRAM but DRAM cannot meet the bandwidth requirements.

For example, with our Sparsey ASIC proposal, the SRAM bandwidth is of the order of 100 terabits per second.

Considering that any neural network specific ASIC might be expected to perform computations for more than one neural network, and that caching will likely prove ineffective, then weights and/or inputs will need to be transferred from “main” memory to local ASIC SRAM at a speed that’s of the order of the SRAM access speed. With current DRAM technology providing maximum bandwidth approaching 1-2Tbsp, there is a bandwidth gap between DRAM and SRAM.

We believe that there is an opportunity to exploit a power and bandwidth benefit from three-dimensional integrated circuit (3DIC) technology. The current bandwidth targets for 3DIC DRAM of 4-8Tbsp do not approach the required bandwidth requirements. We believe there are untapped bandwidth opportunities in 3DIC DRAM. We believe a custom processing layer added to a 3DIC memory stack could improve the raw bandwidth by as little as 8X and potentially much higher. For example, the current Tezzaron DiRAM4 memory has a memory stack sitting on top of a controller layer, which in turn sits on top of an input/output (IO) layer. The IO layer provides the “standard” interface, in this case 64 32-bit interfaces running double data rate (DDR) at 1GTps. The standard interface provides the user access to a portion of a page known as the cache line. We believe that by providing the user access to the entire page would provide an 8X performance boost with limited risk. For example, a low risk redesign of the controller would provide the entire page (4096 bits) rather than a cache line (256 bits) to the layer below. Further redesigns to support simultaneous page open commands will further create a significant performance boost but with increased power. We believe the power increase can be kept within acceptable bounds by limiting the number of simultaneous page open commands with the potential to provide 100’s of terabits per second of usable bandwidth.


Approved for public release; distribution unlimited.

## 7.2 Details

The current IO layer would be a custom-processing layer employing operations that operate directly on data from the DRAM. In many cases, this processing would be performed at line rate and the result written directly back to memory. We have been referring to these operations as PIM or streaming operations and have similarities to the operation we are proposing in our previously mentioned SIMD solution.

The result of the PIM operation will, in some cases be sent to the SIMD array for further processing. In other cases, such as Convolutional neural networks the PIM result can be written directly back to DRAM for later processing by subsequent layers in the network. In this case, a cache may prove useful providing direct access to the PIM result.

An example system with minimal changes to the current DiRAM4 controller layer can be seen in Figure 19. The bandwidth to the SIMD array need only be a small portion of the DRAM bandwidth as it typically only processes the result from the PIM unit.

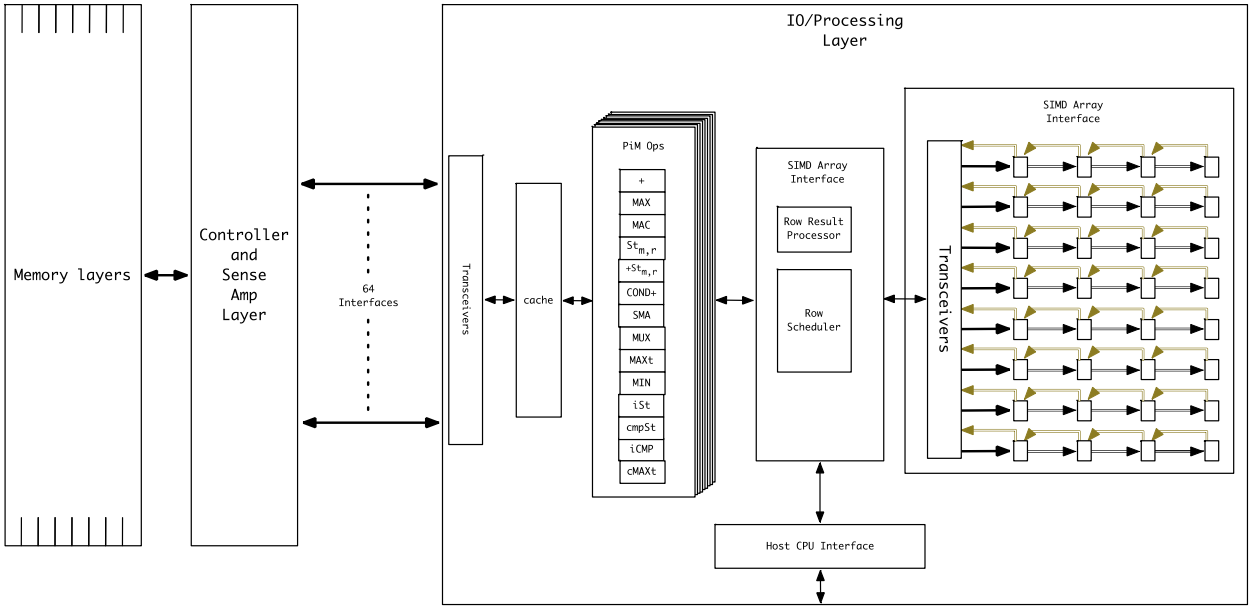


**Figure 19: 3DIC DiRAM4 with additional Processing Layer**


Approved for public release; distribution unlimited.

# 8.0 CONCLUDING REMARKS

These emerging algorithms that can support unsupervised, or lightly supervised learning, as well as incremental learning, map poorly onto conventional computing architectures. The reason is that core to these algorithms, everything is “remembered”, and the resulting reinforced weights cannot be optimized down to a smaller set via off-line optimization.

As a result, this study showed that custom hardware, implemented in the 65 nm node, can improve the power/performance ratio compared with 40 nm/28 nm GPUs by a factor of 100,000 or more. The main reason for this is that the custom solutions have more usable memory bandwidth and are designed to exploit it. Key innovations demonstrated include the use of PIM and the design of parallelized micro-op sequences that permitted faster overall operation.


Approved for public release; distribution unlimited.

# 9.0 REFERENCES

[booksim]: http://nocs.stanford.edu/cgi-bin/trac.cgi/wiki/Resources/BookSim

[fattree]:
http://hpcadvisorycouncil.com/events/2012/EuropeanWorkshop/Presentations/5_Simula.pdf

[Grok]: www.groksolutions.com

[KTH]: http://www.nada.kth.se/cvap/actions/

[mesh]: https://www.ece.cmu.edu/~sld/software/worm_sim.php

# LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

| ACRONYM | DESCRIPTION |
|---|---|
| 3DIC | three-dimensional integrated circuit |
| ASIC | application-specific integrated circuit |
| CM | competitive module |
| CMOS | complementary metal-oxide semiconductor |
| CPU | central processing unit |
| DDR | double data rate |
| DRAM | dynamic random access memory |
| G | global familiarity |
| GF | Global Foundries |
| GPGPU | general-purpose graphic processing unit |
| GPU | graphics processing unit |
| *H* | horizontal |
| HTM | Hierarchical Temporal Memory |
| IO | input/output |
| ISA | instruction set architecture |
| KTH | KTH Royal Institute of Technology |
| MAC/Mac | macrocolumn |
| NoC | network-on-chip |
| PCM | phase change memory |
| PE | processing element |
| PIM | process/processor-in-memory |
| RRAM | resistive random access memory |
| RTL | register transfer language |
| SCM | storage class memory |
| SFU | special function unit |
| SIMD | single instruction multiple data |
| SRAM | static random access memory |
| TLM | transaction level model |
| *U* | bottom-up |
| WTA | winner take all |
| $\eta$ | expansivity |


Approved for public release; distribution unlimited.